ライブプログラミングのためのデータ構造の可視化と対話機能

岡 明央[1,a)]　増原 英彦[2,b)]　青谷 知幸[2,c)]

概要：ライブプログラミング環境はプログラムを編集すると即座にプログラムを実行し，その結果を出力する．これまで，画像や音を出力するプログラムに対して特に有効だと考えられていた．一方，データ構造の定義や操作を行う際，プログラマはオブジェクト間の参照関係を思い浮かべながらプログラムを書くことが多い．そこで我々は，データ構造のためのライブプログラミング環境 Kanon を提案する．具体的には，プログラムが編集される度に，プログラムを実行し，その途中で生成されたオブジェクト及び参照関係 (以下，グラフ構造と呼ぶ) を図表現として可視化する．プログラマによる試行錯誤を容易にするため，以下のような機能を設計・実現した．(1) カーソル位置に連動して実行途中のグラフ構造やその変化を表示する機能，特に複数回実行された命令上にカーソルがある場合は，個別の状態を表示したり，全ての回で起きた変化を一斉に表示する機能，(2) 図表現上のオブジェクトや参照関係をクリックすることで，それらが作られた時のグラフ構造を表示し，コード位置を教えてくれる機能，(3) プログラムの編集によってグラフ構造が変化した際にも，図表現の概形を保つ機能である．JavaScript 言語を対象に，Kanon の処理系を Khan Academy の Live Editor を拡張して作成した．

キーワード：ライブプログラミング，データ構造，可視化

Visualization and Interactive Features for Supporting Data Structures in Live Programming

Akio Oka[1,a)]　Hidehiko Masuhara[2,b)]　Tomoyuki Aotani[2,c)]

Abstract: Live programming environments are tools that, when a piece of a program is edited, instantly execute the program and display its output. They have been recognized as useful to programs that output images and sounds. When the programmers define data structures and their operations, they often write program with references among objects in their mind. We propose a live programming environment, called Kanon, for assisting data structure programming. When a program is edited, it executes the program, collects created object and references among them (hereafter referred to as the graph structure), and displays the graph structure as a node-link diagram. In order to make trial-and-error processes by the programmers easier, we designed and implemented the following mechanisms: (1) a mechanism that displays an intermediate state and changes of the graph structure during the execution in synchronous with the cursor position, including the features for displaying one particular execution at the cursor position or all executions at once, (2) a mechanism, when a node or an edge of the node-link diagram is clicked, that displays the graph structure at the time of creation of the corresponding object or reference and navigates the cursor to the position of the respective code, (3) a mechanism that mostly preserves the shape of the graphical representation even if an edit of the program changes the graph structure. We constructed an implementation of Kanon for JavaScript by extending the Khan Academy's Live Editor.

Keywords: Live Programming, Data Structure, Visualization

1. はじめに

ソフトウェア開発において「開発作業を効率よく行う」ために，様々な研究・開発がなされている．例えばプログラミングを自動化する研究や，開発環境の研究がある．

プログラミング環境の改善の研究の一例に，統合開発環境 (Integrated Development Environment, IDE) がある．IDE とは，プログラミングで用いるエディタやコンパイラ，デバッガ *1 などを統合したプログラム開発環境のことである [17].

IDE のように，プログラミング環境の改善を目的とする研究の一つに，ライブプログラミング環境がある．ライブプログラミング環境 [5] は，プログラムを編集する度に自動的に実行し，その実行結果や実行時情報をユーザに提示する．ライブプログラミング環境については 2.1 節で詳しく説明する．

1.1 本研究の貢献

本研究では，データ構造のためのライブプログラミング環境 Kanon 及びその機能を提案する．Kanon はプログラマがデータ構造を定義・操作する際にそのデータ構造を図表現として可視化し，データ構造の設計や確認を容易にする．ただし，ここでいうデータ構造は連結リストや木構造などを想定している．

Kanon はプログラムが編集される度にプログラムを実行し，その途中で生成されたオブジェクト及び参照関係 (以下，グラフ構造と呼ぶ) をグラフの図表現として可視化する．さらに，編集中のプログラムと図表現との対応をわかりやすくするために以下の機能を備えている．

- 編集時にカーソル位置まで実行した場合のグラフ構造を可視化できる．またそのときプログラム中の変数を表示されているグラフ上に表現できる
- カーソル位置にある命令がもたらす影響を確認できる
- 可視化されたグラフ構造からプログラム内の生成された場所及び文脈に移動できる

2. 背景

2.1 ライブプログラミング環境

ライブプログラミング環境 [5] はプログラムを編集すると即座にプログラムの実行結果や実行時の情報をプログラマに提示する．

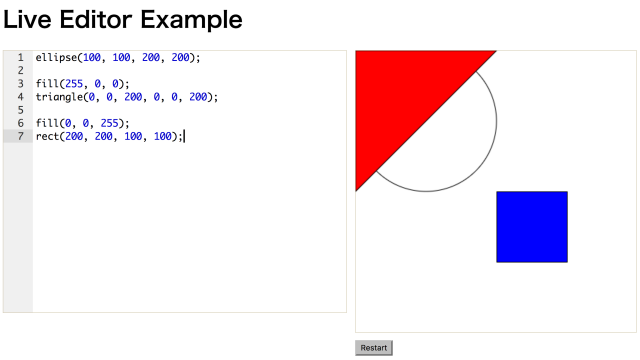

図 1: Khan Academy の Live Editor [15]
Fig. 1 Khan Academy's Live Editor [15].

通常のプログラミングは，「編集，実行，デバッグ」の 3 つのプロセスを繰り返すことで行う．コードを編集したのち，そのコードを実行し，その挙動を観察し，正しい挙動を示すまで「編集，実行，デバッグ」のサイクルを繰り返す．

それに対しライブプログラミングは，「編集，デバッグ」の 2 つのプロセスを繰り返すことで行う．コードを編集すると即座にプログラムの実行結果や実行時情報を提示するため，常に実行結果が正しい挙動を見せているかを確認しながらプログラムを編集することができる．

2.1.1 ライブプログラミングの種類

これまでライブプログラミングは画像を出力するプログラムに特に有効であると考えられていた [1], [8], [15]．しかし近年では数値や文字列を出力するプログラムを対象としたライブプログラミング環境 [6], [7], [10], [14] が増えている．また，現在のライブプログラミング環境には Apple Swift [11] の Playground や，Haskell for Mac [12] のように商品化されているものも存在し，一般的なソフトウェア開発にもライブプログラミング環境が用いられている．

2.1.2 ライブプログラミングの例

2.1.2.1 Live Editor

Khan Academy の Live Editor [15] は JavaScript 及びそのライブラリである Processing.js のライブプログラミング環境である (**図 1**)．Live Editor は左側のエディタでプログラムを編集する．そして，プログラムを編集すると即座にプログラムが実行され，図 1 のエディタに記述されているような描画に関する関数によって描かれる図が右側の表示画面に描画される．

2.1.2.2 YinYang

YinYang [10] は独自のライブプログラミング環境で動作するプログラミング言語・環境で，Probing と Tracing と呼ばれる機能を備えている (**図 2**).

Probing はプログラム中の式の値を表示する機能である．プログラマは式の直前に`@`を挿入すると，その式の値を`@`を挿入した式の直下に表示する．

1 東京工業大学理学部情報科学科
Department of Information Science, Tokyo Institute of Technology
2 東京工業大学情報理工学院数理・計算科学系
Department of Mathematical and Computing Science, Tokyo Institute of Technology
a) a.oka@prg.is.titech.ac.jp
b) masuhara@acm.org
c) aotani@is.titech.ac.jp
*1 デバッグを支援する機能

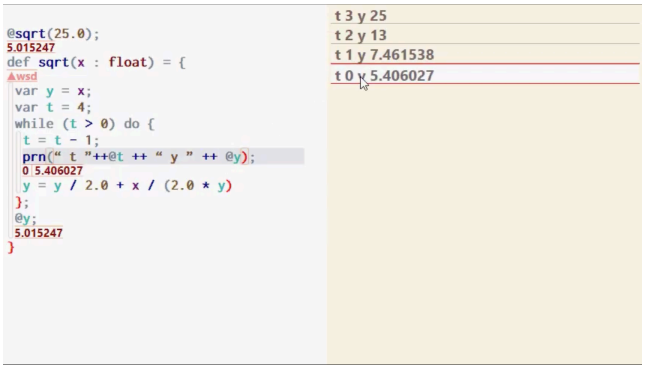

図 2: YinYang の Probing と Tracing [9]

Fig. 2 Probing and Tracing in YinYang [9].

また，Tracing は printf デバッグをサポートするもので，prn 関数の引数として渡された式を評価した値が開発環境の右側に表示される．

Tracing で表示されたトレースをクリックすると，Probing が表示する値をその文脈のものに変化させる．このように，Probing と Tracing を同時に使用することによって，プログラマのデバッグをサポートする．

2.1.3 ライブプログラミングの目標

Tanimoto [16] はライブプログラミングを開発する動機として，“minimizing the latency between a programming action and seeing its effect on program execution” と述べている．また “simplifying the ‘credit assignment problem’ faced by a programmer when some programming actions induce a new runtime behavior (such as a bug)” とも述べており，これらを目標としてライブプログラミングの研究が進められている．

2.2 データ構造の可視化ツール

プログラムを処理する上でデータの集まりを扱うため，一定の形式でデータを格納する．この形式のことをデータ構造という．データ構造には様々な種類があり，扱うデータに適したデータ構造を用いることで，効率よくデータを操作することができる．

記述されたプログラム中で起きるデータ構造の変化をプログラマが容易に確認するためのデータ構造可視化ツールが存在する．データ構造の可視化ツールは数多く存在し，可視化される際に表示される図表現の種類にも様々な表現方法が存在する．

Python Tutor [4] は左側にあるエディタにプログラムを入力するとそのプログラムを解析し，自動的に**図 3** のような図を表示する．この図に表示されている変数やその値は，プログラマが明示的に可視化する対象を選択している訳ではなく，Python Tutor が自動的に解析し，可視化している．また，ステップ実行が可能で，1 つの命令ごとにその時点での図を表示することが可能である．

ghc-vis [3] は Haskell の対話環境で用いるデータ構造可視化ツールで，プログラムを入力するとその命令を実行した後のデータ構造を図表現として可視化する (**図 4**)．ghc-vis は Haskell 特有の遅延評価もグラフ上に表現される．実際，まだ評価されていない部分は図 4 の Thunk のように表現される．

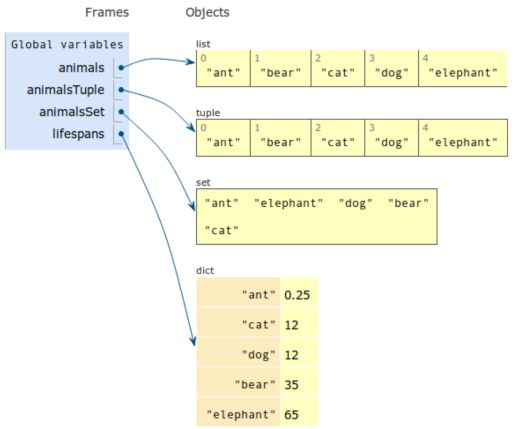

図 3: Python Tutor [4] で表示される図 (出典：Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education [4], 3 ページ目の Figure: 2)

Fig. 3 An example visualization in Python Tutor [4].

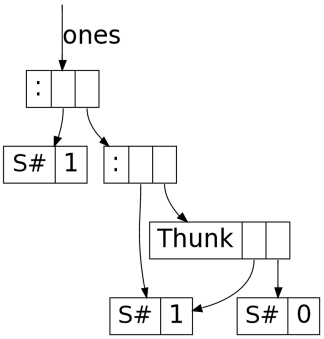

図 4: ghc-vis [3] で表示される図 (出典：`http://felsin9.de/nnis/ghc-vis/`)

Fig. 4 An example visualization in ghc-vis [3].

ghc-vis も Python Tutor 同様，プログラム中で明示的に可視化の対象を選択している訳ではないが，可視化ウィンドウの起動や可視化ウィンドウの更新などは明示的に命令を記述することで行わなければならない．

また，データ構造を操作するアルゴリズムを可視化する Algorithm Visualizer [13] という可視化ツールが存在する．これは，アルゴリズムの動きをアニメーションを用いて可視化する可視化ツールであり，扱うデータ構造の種類に応じて可視化方法が変化する．**図 5** はバブルソートのアルゴリズムを可視化しており，図 5 の中心付近にある棒グラフ (ChartTracer 1) とその下の数字の羅列 (Array 1DTracer 1) が，ソートされる配列を表現している．また，Algorithm Visualizer で表示される図は，可視化する数値やタイミン

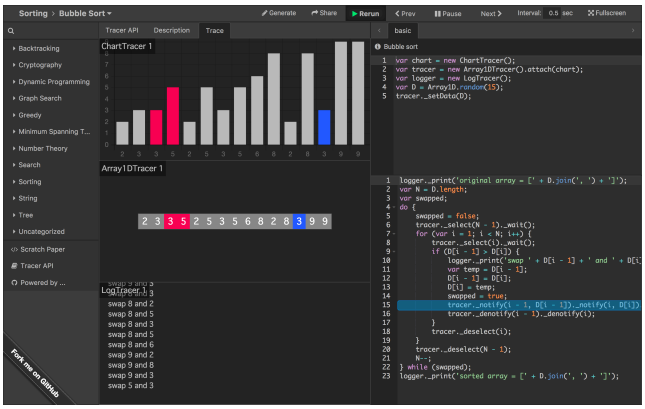

図 5: Algorithm Visualizer [13] のスクリーンショット (バブルソートの例)

Fig. 5 A screenshot of Algorithm Visualizer [13](running a bubble sort algorithm).

グをプログラム中で明示的に記述しなければならない.

3. 問題点・課題

3.1 ライブプログラミング環境の問題点

既存のライブプログラミング環境では，データ構造の扱いが十分ではない．既存のライブプログラミング環境には画像を出力するもの [1], [8], [15] や，数値や文字列を出力するもの [6], [7], [10], [11], [12], [14] などが存在する．しかし，これらのほとんどはデータ構造可視化ツールのような表示をしない *2.

そのため例えば，データ構造に対する操作を記述しているときに操作による変化を観察するためには，プログラマが操作の前後に出力文を挿入しなければならない．さらに，操作による変化が正しいことを確認するためには，出力された文字列から構造を想像した上で変化を読みとらなければならず，プログラマの負担が大きい.

3.2 データ構造可視化ツールの設計上の課題

ツールを設計する上で，どのような可視化方法のツールを作成するのかを考える必要がある．データ構造の可視化方法にも様々な可視化方法が存在する．ここで，可視化方法とは具体的に (1) いつの時点における (2) どのデータ構造を (3) どのような形で (4) どこに表示するか，のような内容を表す.

例えば, Python Tutor [4] は (1) ステップ実行で指定された時点において (2) プログラム中で生成されたオブジェクトを (3) データを長方形の図に格納し，参照関係を矢印で表す表現方法を用いて (図 3)，(4) エディタの隣に表示する，といった可視化方法を選択している．しかし，ghc-vis [3] で可視化されるデータ構造は (1) 対話環境で入力されたプログラムを実行した時点でのデータ構造を (4) 対話環境とは別ウィンドウに表示する.

このように，データ構造可視化には様々な選択肢が存在し，データ構造可視化ツールを設計する上で重要な課題となってくる.

*2 既存のデータ構造を可視化するライブプログラミング環境に関しては 6 章で議論する.

4. Kanon のデータ構造ライブ可視化機能

本研究で提案する Kanon は，データ構造を図表現として可視化するライブプログラミング環境である．しかしデータ構造を可視化する上で，3.2 節で述べた課題を考慮する必要がある．そこで本章では，データ構造可視化における課題の解決策を提示した上で，Kanon の機能を事例を用いて説明する.

4.1 データ構造可視化における課題の解決

3.2 節で述べたように，データ構造を可視化するツールを設計する上でどのような可視化方法を選択するかは一つの課題である．そして本節では，どのような可視化方法を選択したのか，また，選択した可視化方法を実現するために導入した機能を説明する.

Kanon を設計する上で，我々は (1) カーソル位置までプログラムを実行した時点における (2) プログラム上の `new` 式で生成されたオブジェクト及びそのオブジェクトから参照されるオブジェクト全てを (3) オブジェクト名を頂点，オブジェクトのプロパティ名を辺としたグラフの図表現 1 枚で (4) エディタの右に図表現 1 つで表示という選択をした．ただし，プログラムをカーソル位置まで実行した時点のグラフ構造を表示する Snapshot View Mode と呼ばれる機能の他に，プログラムを全て実行した後のグラフ構造を表示し，カーソル位置の変化を表示する Summarized View Mode と呼ばれる機能も実装した．また，表示されるグラフ構造において，リテラルは白い頂点，それ以外のオブジェクトは水色の頂点と，色を変更することで表現した.

実行時情報の数値や文字列を表示するライブプログラミング環境 [6], [7], [10], [14] は各行の命令を実行した際の情報を行の横または付近に表示する．しかし，データ構造を図表現として可視化する Kanon は各行の命令を実行した際の情報を表示するのではなく，カーソル位置を移動することで表示するグラフを変更するという手段を選択した．これは，ユーザに提示する情報が数値や文字列ではなくグラフであるという点から判断した.

4.2 Kanon の機能紹介

本節では，Kanon に備えている機能を紹介する．機能の詳しい説明は 4.3 節及び 4.5 節で説明する.

Kanon に備わっている機能は主に以下の 4 つである.

- Snapshot View Mode
- Summarized View Mode
- Jump to Construction

- Probe

Snapshot View Mode とは，エディタ上のカーソル位置までプログラムを実行した時点でのグラフ構造を表示する機能である．カーソル位置にある命令が複数回実行される場合は，プログラマが指定した文脈におけるグラフ構造を表示する．

Summarized View Mode とは，エディタの命令上にカーソル位置がある時に，その命令で起きた変化をグラフ上に表示する機能である．カーソル位置にある命令が複数回実行される場合は，その命令によって起きた全ての変化を表示する．

Snapshot View Mode 及び Summarized View Mode はプログラム中の命令がデータ構造に及ぼす影響を見せる 2 つの手法 (以下，可視化モードと呼ぶ) であり，プログラマはいずれかを選択して開発を行う．はじめ，可視化モードは Snapshot View Mode が設定されている．

Jump to Construction とは，表示されているグラフの頂点または辺をクリックすると，対応するオブジェクト及び参照関係が生成されたコード位置にカーソルと文脈が移動する機能である．

Probe とは，プログラム上で宣言される変数の先頭に$を挿入すると，その変数が参照しているオブジェクトを指す矢印を表示する機能である．

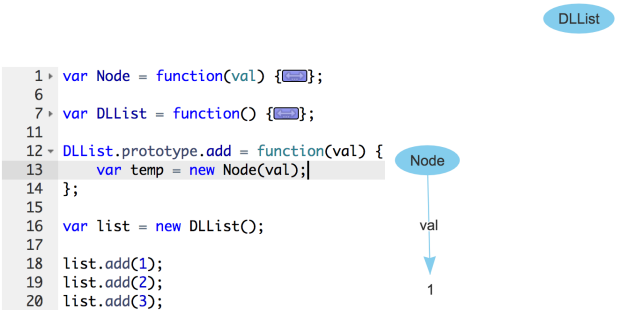

図 7: add メソッドの先頭に Node オブジェクトの生成式を記述した状態

Fig. 7 A screenshot when the first line of the add method of the Node object is written.

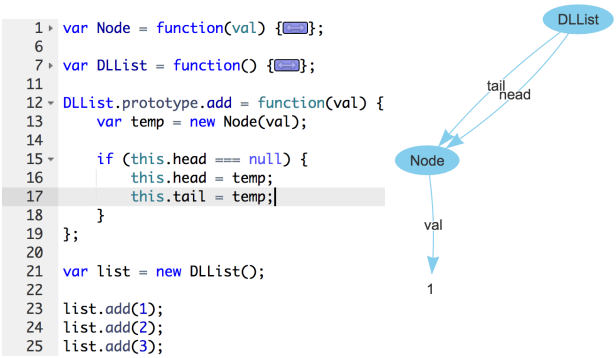

図 8: add メソッドにリストが空の場合のコードを追加した状態

Fig. 8 A screenshot when the code for the empty case is written.

4.3 事例を用いた機能説明

本節では，実際に Kanon を用いて開発を行う一連の流れに沿って Kanon の機能を説明する．本節で用いる事例は双方向連結リストの定義・操作である．

4.3.1 Kanon のユーザインターフェイス

Kanon は，左側にプログラムを入力するエディタ，右側にプログラムを実行し，その途中で生成されたグラフ構造を表示する出力画面がある (**図 6**)．また，エディタの下にある「Summarized」「Snapshot」というボタンは，可視化モードを指定するボタンである．そして，図 6 のボタンの横に書かれている「View Mode: Snapshot」というテキストは，現在選択されている可視化モードを表す．

現在，図 6 のエディタには，以下の 2 つのコンストラクタが定義されている．

- `Node`: 双方向連結リストの各要素を表す．ノードの値と前後のノードの参照を保持する．
- `DLList`: 双方向連結リストを表す．先頭及び末尾のノードの参照を保持する．

図 6 のエディタに記述されているプログラムは機能を説明する上で用いる事例の完成後のプログラムである．また，出力画面に表示されているグラフの水色の楕円はオブジェクト，無地の数値や文字列はリテラルを表示しており，辺の上にあるテキストはプロパティ名を表している．

4.3.2 Snapshot View Mode

Snapshot View Mode の機能を `DLList` コンストラクタに `add` メソッドを定義する場面を用いて説明する．`add` メソッドは引数を 1 つ受け取り，その値を `val` プロパティに保存した `Node` オブジェクトをリストの最後尾に追加する．

まずプログラマは，`add` メソッドの外枠及び複数のメソッド呼び出しを記述し，`add` メソッドの本体の 1 行目を記述する (**図 7**)．

ここで出力画面には図 7 の 18 行目のメソッド呼び出し文脈内の 13 行目のカーソル位置まで実行した時点でのグラフ構造が表示されている．Snapshot View Mode はカーソル位置まで実行した時点でのグラフ構造を表示するモードであるが，複数回実行された処理上にカーソル位置がある場合，プログラマは「何周目のカーソル位置まで実行した時点のグラフ構造を可視化するか (以下，可視化文脈と呼ぶ)」を指定できる．可視化文脈の初期値は 1 周目であるため，`add` メソッドの 1 周目のカーソル位置まで実行した時点のグラフ構造が表示されている．

この表示されたグラフを確認しながらコードを記述する．`list.add(1)` のメソッド呼び出し時のカーソル位置まで実行した時点のグラフ構造が表示されることを利用して，リストが空の場合を記述する (**図 8**)．その際，直前に記述した処理が出力画面のグラフに反映されるため，プログラマは記述したプログラムの挙動を確認することができる．

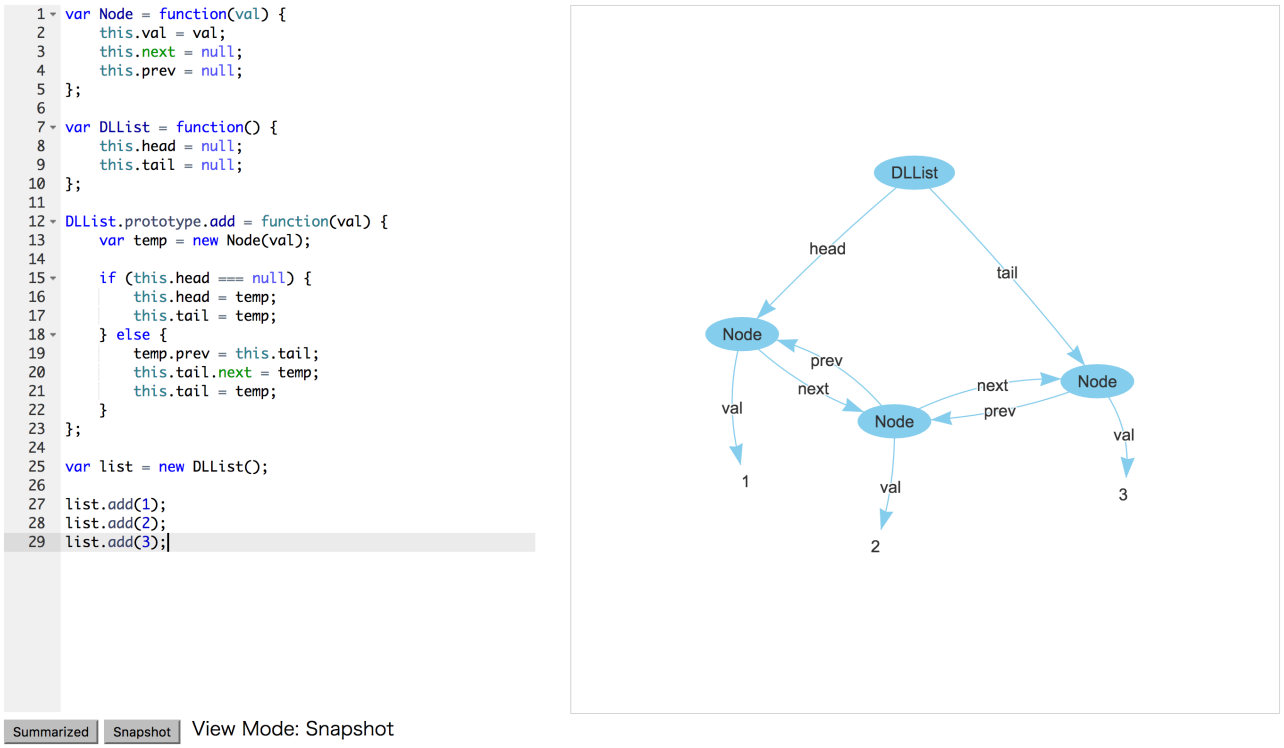

図 6: Kanon の実際の画面

Fig. 6 A screenshot of Kanon.

4.3.3 Probe

ここでは，add メソッド内の if 文の else 節を記述していく．ただし，ここでは Snapshot View Mode の可視化文脈が 2 周目 (list.add(2) のメソッド呼び出し文脈) に指定されている．可視化文脈の指定は Jump to Construction を用いるが，指定方法は 4.4 小節で述べる．

else 節を記述する上で，変数 temp がどのオブジェクトを参照しているかを理解する必要がある．そこで，13 行目で宣言されている変数 temp の変数名の先頭に$を挿入する (**図 9** の 13 行目)．するとグラフ上に temp とかかれた矢印が図 9 のように追加される．これは変数 temp が val プロパティの値が 2 の Node オブジェクトを参照していることを表している．これによりプログラマは変数 temp が参照しているオブジェクトを確認することができ，記述がより容易になる．そして，add メソッドの else 節を記述する (図 9)．

4.3.4 Summarized View Mode

add メソッドの本体を記述し終えたため，記述したプログラムが正しい挙動を示しているか確認をする．ここでは，プログラムを全て実行した時点の構造を確認する．現在，Snapshot View Mode に指定されているので，Kanon のエディタの下方にある「Summarized」と書かれているボタンをクリックし，Summarized View Mode に変更する．

Summarized View Mode に変更すると，プログラムを全て実行した時点のグラフ構造が表示される (**図 10**)．双方向連結リストは，prev プロパティと next プロパティが対になっているはずであるが，図 10 では対になっておらず，

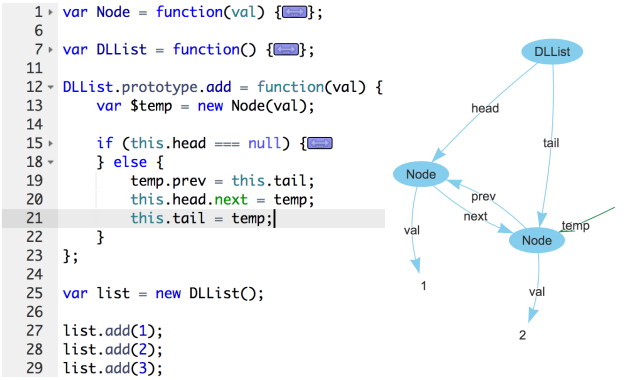

図 9: add メソッドのリストが空でない場合 (else 節) を記述した時の Probe の働き

Fig. 9 Visualization of a Probe when the code for the non-empty case is written.

add メソッドが期待通り動作していないことが確認できる．

そこでプログラムのバグを探すため，Summarized View Mode の状態でカーソルを移動させる．Summarized View Mode では，カーソル位置の命令においてグラフ構造がどのように変化したのかを確認することができる．カーソルを図 9 の 19 行目の命令 (temp.prev = this.tail;) にカーソルを移動する．この時，**図 11** のようなグラフが表示され，prev の辺がオレンジ色に変化しているのがわかる．これは，カーソル位置にある命令でグラフ構造に prev の辺 (参照関係) が追加されたことを表している．これによりプログラマは 19 行目の命令が正しく動作していると判断できる．

続いて，図 9 の 20 行目の命令 (this.head.next =

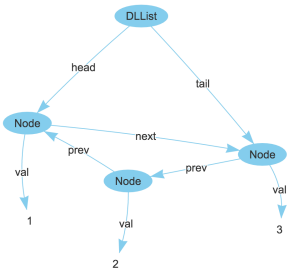

図 10: 図 9 のプログラムで Summarized View Mode の時のグラフ表示

Fig. 10 A summarized view for the program in Fig. 9.

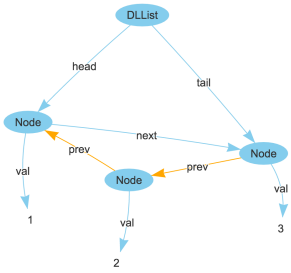

図 11: 図 9 の 19 行目 (`temp.prev = this.tail;`) にカーソルが存在する場合

Fig. 11 A summarized view for Fig. 9 with the cursor at line 19.

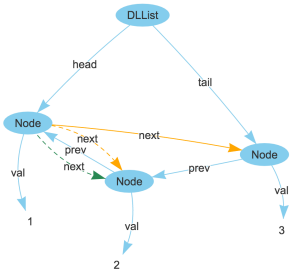

図 12: 図 9 の 20 行目 (`this.head.next = temp;`) にカーソルが存在する場合

Fig. 12 A summarized view for Fig. 9 with the cursor at line 20.

`temp;`) にカーソルを移動する．すると，この命令が加えたグラフ構造への変化が**図 12** のグラフ上に表示される．ここで，破線の辺はプログラムを全て実行した時点で存在しない参照関係を表し，緑色の辺は，現在のカーソル位置の命令で消去された参照関係を表している．そのため，プログラマは図 12 を見ることで，20 行目の命令では (1) 初めは `val` の値が 1 の `Node` オブジェクトから 2 の `Node` オブジェクトに `next` の辺が期待通りに存在していたということと，(2) 一度は期待通りに存在していた `next` の辺を `val` の値が 3 の `Node` オブジェクトに移動してしまっているということが読み取れる．

よって，図 9 のプログラムの 20 行目の命令 (`this.head.next = temp;`) は，`this.head` の `next` プロパティではなく，`this.tail` の `next` プロパティに変数 `temp` の値を代入しなければならないとわかる．そこで，`this.head.next = temp;` のうちの `head` という部分を `tail` と書き換えると，**図 13** のようなグラフ表示がされ，記述したプログラムが正しい挙動をしていることが即座にわかる．

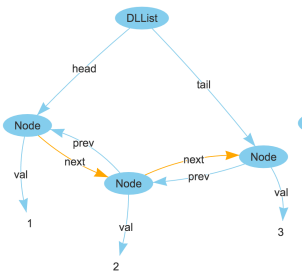

図 13: 図 6 の 20 行目 (`this.tail.next = temp;`) にカーソルが存在する場合

Fig. 13 A summarized view for Fig. 6 with the cursor at line 20.

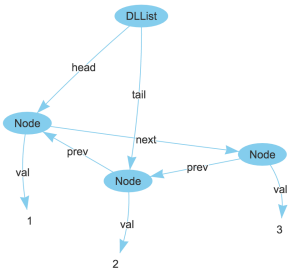

図 14: 図 10 の `next` の辺をクリックした後のグラフ表示

Fig. 14 A graph view after the programmer clicks the `next` edge in Fig. 10.

4.4 Jump to Construction

4.3.4 小節では，カーソルを移動させて各命令の変化を見ることでバグの原因を特定していた．ここでは，バグの原因を特定するもう一つの方法について述べる．

Summarized View Mode に変更して図 10 のグラフが表示されたとき，グラフの `next` の辺が期待する位置に存在していないということがわかる．そこで，この `next` の辺が生成された場所を探すために，グラフの `next` の辺をクリックする．すると，(1) カーソルが図 9 の 20 行目 (`this.head.next = temp;`) の直後に移動し，(2) 可視化モードが Snapshot View Mode に変更され，(3) `add` メソッドの可視化文脈が，クリックした `next` の辺が生成された文脈に変更される．そのため，グラフ表示は**図 14** のようになる．この Jump to Construction を用いても，プログラマはバグの原因の特定が可能である．

4.5 機能の詳細

4.3 節では，実際に開発する手順に沿って Kanon の機能を説明した．本節では，4.3 節で登場しなかった場面も含め，それぞれの機能の詳しい説明をする．

4.5.1 Snapshot View Mode

Snapshot View Mode は可視化モードの一種で，エディタのカーソル位置までプログラムを実行した時点でのグラフ構造を表示する．また，カーソル位置が複数回実行される処理の上にある場合はプログラマが可視化文脈を指定することで何周目のカーソル位置まで実行した時点でのグラフ構造を表示するかを設定することができる．

Snapshot View Mode の時，表示されるグラフは以下の情報から決定される．

- カーソルの位置
- カーソル位置のループの可視化文脈

ここでいうループとは，関数の本体や `for` 文，`while` 文といった「複数回実行される可能性のある命令列」を指す．可視化文脈はループごとに内部的に保存されており，複数

ソースコード 1: 複数のループが存在する場合

```
1  var f = function() {
2      // 命令 1
3  };
4
5  var g = function() {
6      // 命令 2
7  };
8
9  f(); f(); f();
10 g(); g();
```

ソースコード 2: 実行されない処理を含むプログラム

```
1  if (true) {
2      // 命令 1
3  } else {
4      // 命令 2
5  }
```

のループが存在した場合には，ループごとに可視化文脈が保存されている．例えば，ソースコード 1 の場合では関数 f は 3 回，関数 g は 2 回呼び出されている．この時，関数 f の可視化文脈を 2 周目，関数 g の可視化文脈を 1 周目といったように指定できる．その場合，カーソル位置が命令 1 にある場合は関数 f の 2 周目のカーソル位置まで実行した結果を表示し，カーソル位置が命令 2 にある場合は関数 g の 1 周目のカーソル位置まで実行した結果を表示する．

また，カーソル位置まで実行した時点でのグラフ構造を表示するため，カーソル位置が処理されない位置に存在する場合は出力画面に何も表示されない．例えばソースコード 2 のように条件分岐の場合，4 行目の命令 2 は処理されない命令となる．この時に命令 2 にカーソルが位置していると，この if 文に到達する時点でグラフ構造が定義されていたとしても出力画面には何も表示されない．

4.5.2 Jump to Construction

Jump to Construction は，クリックした頂点または辺が表すオブジェクト及び参照関係が生成されたコード位置にカーソルが移動する機能である．

出力画面に表示されているグラフの頂点または辺をクリックした際，コード位置にカーソルが移動する．このとき，クリック時の可視化モードに関わらず，クリック後の可視化モードは必ず Snapshot View Mode となる．可視化モードを Snapshot View Mode にし，可視化文脈を変更することで，クリックされた頂点または辺が表すオブジェクトまたは参照関係が生成された文脈をプログラマに提示しているのである．

また，Probe によって表示された変数を表す矢印や Summarized View Mode の時に表示される破線の辺も Jump to Construction の対象である．そのため，Probe による矢印をクリックした場合はその変数が宣言されたコード位置に移動し，Summarized View Mode の時に表示される破線の辺をクリックした場合は，その破線の辺が表す参照関係が生成されたコード位置に移動する．

4.5.3 Probe

Probe とは，プログラムをカーソル位置まで実行した時点において，変数が参照しているオブジェクトを指す矢印をグラフ上に表示する機能である．Probe による矢印は Snapshot View Mode の時にのみ表示され，その矢印は緑色である．

Probe は，変数宣言で宣言する変数名の先頭に$を挿入することでその変数を Probe の対象とする．Probe の対象となった変数は，$が挿入される前の変数名として動作し，グラフ上に表示される (図 9 の変数 temp)．

また，変数にはスコープが存在する．Probe は変数の参照を表示するため，Probe の対象となっている変数のスコープ外にカーソルがある場合は，その変数の参照を表す矢印は表示されない．

4.5.4 Summarized View Mode

Summarized View Mode は可視化モードの一種で，可視化されるグラフ構造はプログラムを全て実行した後のオブジェクト及びその参照関係である．そして，カーソル位置における命令がもたらした変化を，グラフ上に表示する可視化方法である．

カーソル位置における命令がもたらした変化は，グラフの「頂点・辺の色」及び「辺の種類」を変更することで表現される．そして「頂点・辺の色」及び「辺の種類」は以下の内容を表している．

- 頂点・辺の色：カーソル位置の命令によって追加または削除されたことを表す
  - オレンジ：カーソル位置の命令で**追加**されている
  - 緑：カーソル位置の命令で**削除**されている (辺のみ)
  - 水色：カーソル位置の命令では変化していない
- 辺の種類：プログラム実行終了時に存在した参照関係かどうかを表す
  - 実線：終了時に存在する
  - 破線：終了時に存在しない

4.6 その他の機能

前節で説明した機能以外に，開発を行う上で便利な機能が備わっている．そこで本節では，前節で説明した機能以外で Kanon に備わっている機能を紹介する．

4.6.1 無限ループ対策

Kanon のエディタで無限ループを起こすプログラムを記述すると，可視化モードを選択するボタンの下方に「infinite loop?」と表示される．「infinite loop?」と表示されている間はカーソルを移動させても表示されているグラフは全く

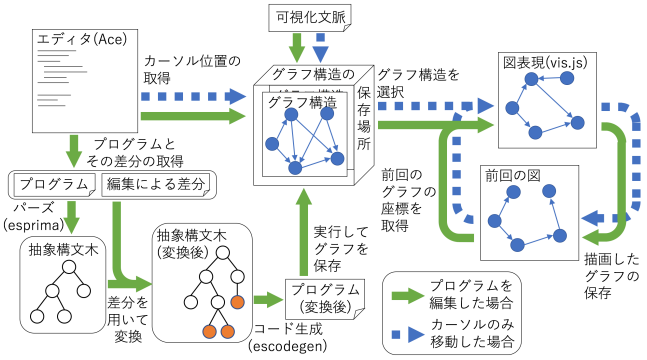

図 15: 全体の流れ

Fig. 15 Overview of the implementation.

変更されず，Jump to Construction なども停止する．

5. Kanon の実現方法

本章では，Kanon の処理系及びその機能を実現する方法を述べる．また，実装する上で工夫した点を述べる．

Kanon のソースコードは 72 行の HTML と 28 行の CSS，約 1,900 行の JavaScript で構成されており，外部ライブラリとして Ace[*3]，esprima[*4]，escodegen[*5]，vis.js[*6] を用いている．また，Kanon はオープンソースのライブプログラミング環境として，`https://github.com/prg-titech/Kanon` において公開されている．

5.1 全体の流れ

本節では，プログラマが Kanon のエディタ上でプログラムを編集またはエディタのカーソルを移動してから，出力画面にグラフ構造が表示されるまでの全体の流れ (**図 15**) に沿って，Kanon がどのように実現されているのかを説明する．

5.1.1 エディタからプログラムを取得

Kanon のエディタには，Ace と呼ばれる，JavaScript で実装されていて Web 上で用いる埋め込み型エディタを用いている．

Kanon のエディタを定義する．ソースコード 3 の 1 行目で`__$__`という変数がグローバルに定義されている．変数`__$__`については 5.1.3 小節で説明する．

エディタはソースコード 3 の 2 行目で定義され，`__$__.editor` の初期値として与える．そしてエディタからプログラムを取得する際，エディタに対して `getValue()` メソッドを呼び出す (ソースコード 3 の 4 行目)．

5.1.2 プログラム変換

あるプログラムを元に，別のプログラムに書き換えることをプログラム変換という．本小節では，図 15 における「プログラム」から「プログラム (変換後)」までのプログラム変換を実現する方法について説明する．そしてプログラマが記述したプログラムを変換する際の変換規則及び変換後のプログラムの役割について説明する．

*3 `https://ace.c9.io`
*4 `http://esprima.org/`
*5 `https://github.com/estools/escodegen`
*6 `http://visjs.org/`

ソースコード 3: エディタの定義とプログラムの取得

```
1 var __$__ = {};
2 __$__.editor = ace.edit("editor");
3
4 __$__.editor.getValue();
```

ソースコード 4: プログラムの先頭で変数宣言

```
1 var __newIdCounter = {};
2 var __objs = [];
3 var __loopId = "noLoop";
4 var __loopCount = 1;
5 var __loopCounter = {};
6 var __time_counter = 0;
```

5.1.2.1 変換手段

プログラム変換をするためには，まず一度プログラムを抽象構文木に変換する．そして，その抽象構文木を規則に従って変換し，変換された抽象構文木から変換後のプログラムを生成することで実現する．

プログラムを抽象構文木に変換するために，我々は esprima と呼ばれる JavaScript 用のパーザを用いた．また，抽象構文木からプログラムを作成するために，escodegen と呼ばれる JavaScript 用のコードジェネレータを用いた．

抽象構文木の変換は，深さ優先探索で走査を行うことで実現する．深さ優先探索で走査を行うプログラムは，Live Editor [15] の `ast-walker.js`[*7] というプログラムを参考に作成した．

5.1.2.2 変換内容

プログラム変換をする目的は主に以下の 4 つである．

- プログラム上で用いる変数の定義
- プログラム内で生成されたオブジェクトの収集
- ループ及び関数の管理
- チェックポイントの挿入

以下では，これらを実現する方法及び役割を説明する．

5.1.2.2.1 変数の定義

プログラム変換において，プログラムの先頭で変数を宣言する．宣言される変数はソースコード 4 に記す．詳しくはこれらの変数を用いる際に説明する．

5.1.2.2.2 オブジェクトの収集

プログラマが記述したプログラム中で生成されたオブジェクトを収集する．プログラム中の `new` 式を無名関数の呼び出しに変換することで実現した (ソースコード 5)．

*7 `https://github.com/Khan/live-editor/blob/master/js/output/shared/ast-walker.js`

ソースコード 5: new 式の変換

```
1  // 変換前
2  new Class(arg1, arg2, ...)
3
4  // 変換後
5  (() => {
6      var temp = new Class(arg1, arg2, ...);
7
8      if (__newIdCounter[固有のID]) __newIdCounter[固有のID
           ]++;
9      else __newIdCounter[固有のID] = 1;
10
11     temp.__id = 固有のID + __newIdConter[id];
12     __objs.push(temp);
13     return temp;
14 })()
```

ソースコード 6: ループまたは関数の変換

```
1  // 変換前（本体がブロックで囲われていない場合）
2  while (condition)
3      ...
4
5  // 変換前（本体がブロックで囲われている場合）
6  while (condition) {
7      ...
8  }
9
10 // 変換後
11 while (condition) {
12     let __loopId = "loop2";
13     if (!__$__.Context.LoopContext[__loopId])
14         __$__.Context.LoopContext[__loopId] = 1;
15     let __loopCount = (__loopCounter[__loopId])
16         ? ++__loopCounter[__loopId]
17         : __loopCounter[__loopId] = 1;
18     if (__loopCount > 10000)
19         throw "Infinite Loop";
20     let __start = __time_counter;
21     ...
22     if (!__$__.Context.StartEndInLoop[__loopId])
23         __$__.Context.StartEndInLoop[__loopId] = [];
24     __$__.Context.StartEndInLoop[__loopId].push(
25         {start: __start, end: __time_counter - 1});
26 }
```

抽象構文木の変換を行う際，プログラムに記述されている全ての new 式に固有の ID を"new1"，"new2"，... と言う形でる．ソースコード 5 中の変数__newIdCounter 及び __objs は変換後のプログラムの先頭で宣言され (ソースコード 4 の 1, 2 行目)，__newIdCounter はプログラム中のそれぞれの new 式が呼び出された回数を保存するオブジェクトであり，__objs はプログラム中の new 式で生成されたオブジェクトの参照を保存する配列である．

5.1.2.2.3 ループ及び関数の管理

ループ及び関数は，プログラムを実行する際に複数回処理される可能性を含む．抽象構文木を変換する上で，その可能性を含むのは，以下の 7 つのノードの場合である．

- WhileStatement
- DoWhileStatement
- ForStatement
- ForInStatement
- FunctionDeclaration
- FunctionExpression
- ArrowFunctionExpression

走査中にこれらのノードのいずれかに遭遇した際，ソースコード 6 のような変換を行う．ソースコード 6 は，WhileStatement の例である．ただし，変換前のノードの本体がブロックでない場合 (ソースコード 6 の 2 行目) は，先にブロックで本体を囲ってから，プログラムの挿入を開始する (ソースコード 6 の 6 行目)．

12 行目では，ループの固有の ID を__loopId という変数名で宣言している．ID は，"loop1"，"loop2"，... という形でつけられる．また，プログラムの先頭で変数__loopId を宣言し，初期値が"noLoop"であるため，ループ外であっても変数__loopId は有効である (ソースコード 4 の 3 行目)．また，let を用いて宣言された変数はスコープがブロック内に制限される．

13 行目は Snapshot View Mode の際の可視化文脈の初期化を行っている．ここで，__$__.Context.LoopContext は可視化文脈を保存するオブジェクトである．

15 行目では，変数__loopCount を宣言している．変数 __loopCount 及び__loopCounter はプログラムの先頭で宣言されており，__loopCount は現在のループの周回数を保存する変数，__loopCounter は各ループが実行された回数を管理しているオブジェクトである (ソースコード 4 の 4, 5 行目)．

18 行目では，無限ループを回避するために，このループの処理回数が 10000 回を超えた場合はエラーを返す．

20 行目及び，22 行目から 25 行目において，このループがプログラムを実行する上でどのタイミングで処理されているのかを管理している．

変数__time_counter は，プログラムを実行する上での時間軸を管理する変数で，25 行目でループの始まるタイミングを変数__start に保存し，27 行目から 30 行目で __$__.Context.StartEndInLoop という各ループの開始及び終了のタイミングを管理するオブジェクトに現在のループの開始及び終了のタイミングを保存している．

__$__.Context.StartEndInLoop というオブジェクトにおいて，ループの ID が"noLoop"の場合はプログラムの先頭で定義され，プログラムの最後で終了のタイミングを保存している (ソースコード 7)．

5.1.2.2.4 チェックポイントの挿入

表示されるグラフはカーソル位置に依存するため，プロ

ソースコード 7: プログラムの先頭及び最後で"noLoop"の開始，終了タイミングを保存

```
__$__.Context.StartEndInLoop["noLoop"] = [{start: 0}];

...

__$__.Context.StartEndInLoop["noLoop"][0].end =
    __time_counter;
```

グラム中の各命令がグラフ構造に与える影響を管理する必要がある．そこで，プログラムの各文の前後にチェックポイントを設ける．

チェックポイントは，__$__.Context.CheckPoint() という関数を挿入することで実現され，この関数の実引数は以下の 6 つである．

- 第 1 引数: __objs
- 第 2 引数: __loopId
- 第 3 引数: __loopCount
- 第 4 引数: __time_counter++
- 第 5 引数: チェックポイントの ID(整数型)
- 第 6 引数: 可視化される変数の情報を持つオブジェクト

ここで第 4 引数からわかるように，チェックポイントを通過するたびに変数__time_counter の値は 1 ずつ増加する．

第 5 引数のチェックポイントの ID は，全てのチェックポイントに対して固有の ID がつけられる．これは，抽象構文木の走査中に挿入されたチェックポイントの順番である．

第 6 引数は，Probe の対象である変数名及びその変数の値を保存しているオブジェクトである．抽象構文木を深さ優先探索で走査しているため，抽象構文木のノードで先頭に$が挿入された変数宣言に遭遇した場合は変数の先頭に挿入されている$を取り除き，Probe の対象となる変数のスコープ内に挿入されるチェックポイントの第 6 引数のオブジェクトに変数名及びその変数の値を追加する．可視化する変数のスコープ管理については 5.3.3 小節で述べる．

チェックポイントを挿入する際，プログラムのどの位置にどの ID のチェックポイントが挿入されるのかを保存する．詳しくは 5.1.4 小節で述べる．

チェックポイントの挿入は，主にソースコード 8 のような変換を行うことで実現する．ここで，関数 __$__.Context.CheckPoint() の引数は省略してある．ただしチェックポイントの挿入方法には文の種類によって一部例外が存在する．以下では，その例外について述べる．

1 つ目の例外は，以下の 3 つの文の場合である．

- ReturnStatement
- ContinueStatement

ソースコード 8: 基本的なチェックポイントの挿入方法

```
// 変換前
Statement;

// 変換後
{
    __$__.Context.CheckPoint(...);
    Statement;
    __$__.Context.CheckPoint(...);
}
```

ソースコード 9: 直前にのみチェックポイントを挿入する場合の例

```
// 変換前
return;

// 変換後
{
    __$__.Context.CheckPoint(...);
    return;
}
```

ソースコード 10: ブロックを用いずにチェックポイントを挿入する場合の例

```
// 変換前
...
let v;
...

// 変換後
...
__$__.Context.CheckPoint(...);
let v;
__$__.Context.CheckPoint(...);
...
```

- BreakStatement

これらの場合，この文の直後は処理されないため，チェックポイントは文の直前にのみ挿入する (ソースコード 9)．

2 つ目の例外は，let，const を用いた変数宣言 (VariableDeclaration) の場合である．JavaScript において，let または const を用いた変数宣言は，スコープがブロック内に制限されるため，ソースコード 8 の変換では変換前と同じ挙動を示さない．そのため，この場合は変換前の文をソースコード 10 のように変更することで実現する．

ただし，変数宣言 (VariableDeclaration) の例外として，for 文の初期化部分における変数宣言があげられる．for 文の初期化部分での変数宣言は宣言の種類に関わらずチェックポイントを挿入しない．

ソースコード 11: 変数の初期化の前後にチェックポイントを挿入する場合

```
// 変換前
var v = [expression], v2 = [expression];

// 変換後
var v = (() => {
    __$__.Context.CheckPoint(...);
    var v = [expression];
    __$__.Context.CheckPoint(...);
    return v;
})(), v2 = (() => {
    __$__.Context.CheckPoint(...);
    var v2 = [expression];
    __$__.Context.CheckPoint(...);
    return v2;
})();
```

3 つ目の例外は，ノードが VariableDeclarator の場合である．通常，チェックポイントは各文の前後に挿入している．しかし，同時に複数の変数が宣言された場合，各変数の初期化部分で行われている処理を管理できない．そこで，それぞれの変数宣言の初期化部分の前後にチェックポイントを挿入する．これは，変数宣言の初期化部分を無名関数の呼び出し式に変換することで，初期化部分の前後へのチェックポイントの挿入を実現した (ソースコード 11)．

5.1.3 変換されたプログラムの評価

変換されたプログラムは，JavaScript の eval() という関数を用いて評価される．この処理は，図 15 の「プログラム (変換後)」から「グラフ構造の保存場所」の部分に対応する．

eval() 関数は，呼び出された環境での変数束縛を用いてプログラムを評価するため，プログラマが記述したプログラム上での変数名と，実装上で用いる変数名の衝突が考えられる．

この問題に対する解決として，実装上で定義されている変数は__$__という名前のオブジェクトのみにし，実装上で用いる関数や変数を全て__$__オブジェクトのプロパティとした．プログラマが__$__という名前の変数を用いた場合は変数名の衝突が生じるが，__$__という名前の変数が用いられるケースは滅多にないと考えられるため，この解決方法で十分であると判断した．

5.1.3.1 チェックポイントの処理

ここでは，eval() 関数を用いて変換されたプログラムを評価する際に，チェックポイント内で行われる処理について述べる．

チェックポイントでは，以下の処理が行われる．

- グラフ構造を保存する
- 頂点または辺が初めて現れた場所を調べる (5.2 節)
- 多重ループの管理 (5.3.4 小節)

ソースコード 12: グラフ構造のコンストラクタ

```
__$__.Traverse.__Graph = function() {
    this.nodes = [];
    this.edges = [];
};
```

ソースコード 13: グラフ構造のリテラルのコンストラクタ

```
__$__.Traverse.__Literal = function(value) {
    this.value = value;
};
```

チェックポイントでは，そこに到達した時点のグラフ構造を__$__.Context.StoredGraph というオブジェクトに以下の情報を付加して保存する．

- チェックポイントの ID
- 現在のループの ID
- ループ回数

チェックポイントの第一仮引数には，変換されたプログラム中の変数__objs の参照が代入されている．変数__objs の参照を直接保存した場合，変数__objs が参照するオブジェクトのプロパティが変更される可能性が生じる．そのため，チェックポイントに到達した時点のグラフ構造を保存するには，変数__objs に代入されている参照を保存するのではなく，__objs のグラフ構造と同等のオブジェクトを生成し，保存する必要がある．

また，__objs に保存されるオブジェクトは new 式を用いて生成されたオブジェクトのみであり，new 式を用いずに生成されたオブジェクトやリテラルは保存されていない．

そこで，保存するグラフ構造を生成するために，チェックポイントに到達した時点でのグラフ構造を深さ優先探索を用いて走査する．走査によって頂点及び辺の配列をプロパティとして持つオブジェクトが生成される．そして Kanon では，JavaScript のグラフ可視化ライブラリである vis.js を表示するグラフに用いているため，生成されたオブジェクトを vis.js に適したフォーマットに変換してグラフ構造を保存する．

5.1.3.1.1 グラフ構造の走査

深さ優先探索を用いてグラフ構造の走査を行う．図表現として可視化するために，変換されたプログラム中の__objs から頂点及び辺の配列を持つグラフ構造 (ソースコード 12) を生成する．

ソースコード 12 の nodes プロパティには，__objs から辿ることのできる全てのオブジェクト及びリテラルが保存される．ただし，リテラルは値をプロパティに持つオブジェクトとして，ソースコード 13 のコンストラクタから生成される．

ソースコード 14: グラフ構造の辺のコンストラクタ

```
1 __$__.Traverse.__Edge = function(from, to, label) {
2     this.from = from;
3     this.to = to;
4     this.label = label;
5 };
```

そしてソースコード 12 の edges プロパティには，nodes プロパティのオブジェクト間の参照関係がソースコード 14 のコンストラクタで定義されたオブジェクトで保存される．ここで，from，to プロパティにはオブジェクトの参照を保存し，label プロパティにはグラフ構造のプロパティ名を文字列型で保存する．

また，__$__.Traverse.__Graph の nodes プロパティに保存されている全てのオブジェクトは，走査の際に__id プロパティが定義される．__id プロパティとは，そのオブジェクトを表す固有の ID である．5.1.2.2.2 段落で述べたように，new 式で生成されたオブジェクトには既に__id プロパティが定義されている．しかし，走査で発見されたオブジェクトには__id プロパティは定義されていない．

ソースコード 15 は，__id プロパティを定義する関数 CheckId である．関数 CheckId の引数は，以下の 2 つである．

- node ： __id プロパティが定義されていないオブジェクト
- edges ： __$__.Traverse.__Graph コンストラクタで生成されたオブジェクトの edges プロパティ

グラフ構造の走査を開始するオブジェクトは全て new 式で生成されたオブジェクトなので__id プロパティが定義されている．また，グラフ構造の走査はオブジェクトのプロパティを辿っている．そのため，走査によって発見されたオブジェクトは全て，new 式で生成されたオブジェクトから参照を繰り返すことで到達可能である．この事実を用いると，走査によって発見されたオブジェクトは，そのオブジェクトを参照していてかつ__id プロパティが定義されているオブジェクトから，__id プロパティが定義できる (ソースコード 15 の 7 行目)．

5.1.3.1.2 Probe の対象になった変数の追加

Probe を実現するために，対象となる変数の数だけ透明な頂点を設定し，その透明な頂点から Probe の対象の変数の値の頂点まで矢印を表示するようにした．

透明な頂点は，ソースコード 16 のコンストラクタで定義され，__$__.Traverse.__Graph コンストラクタで定義されたオブジェクトの nodes プロパティに追加される．ここで，ソースコード 16 のコンストラクタの引数 id には，Probe の対象の変数名を渡す．

また，Probe の対象の変数を表す矢印は

ソースコード 15: __id プロパティを定義する関数 CheckId

```
1  __$__.Traverse.CheckId = function(node, edges) {
2      for (var i = 0; i < edges.length; i++) {
3          if (node == edges[i].to) {
4              if (!edges[i].from.__id)
5                  __$__.Traverse.CheckId(edges[i].from,
                       edges);
6
7              node.__id = edges[i].from.__id + "-" + edges
                   [i].label;
8              return;
9          }
10     }
11 };
```

ソースコード 16: グラフ構造の透明な頂点のコンストラクタ

```
1 __$__.Traverse.__VariableNode = function(id) {
2     this.id = id;
3     this.__id = "__Variable-" + id;
4 };
```

__$__.Traverse.__Edge コンストラクタで定義され (ソースコード 14)，from プロパティに透明な頂点を，to プロパティに Probe の対象の変数が参照するオブジェクトを，label プロパティに変数名を代入する．

5.1.3.1.3 vis.js への変換

vis.js ではグラフの各頂点に ID を設定し，辺は頂点の ID を 2 つ指定することで定義する．そこで，グラフ構造の各オブジェクトに定義されている__id プロパティの値をグラフの頂点の ID として用いる．

頂点の名前はオブジェクトのコンストラクタ名を表示するよう指定する．しかし，リテラルの場合はその値を直接表示し，リテラルの頂点の色は"white"に指定している．

Probe の対象の変数を表す矢印の色は"seagreen"に指定している．ここで，辺の from プロパティが参照しているオブジェクトの__id プロパティが"__Variable-"で始まっているかどうかで Probe による矢印であるかを判断する．また，透明な頂点は hidden プロパティを true にすることで実現している．

5.1.4 可視化するグラフの選択

表示されるグラフは，可視化モードによって変化する．ここでは，可視化モードに関わらず，エディタのカーソルの位置に応じてグラフが変化することから，カーソルの位置に応じたチェックポイントの選択方法について説明する．そして，表示されるグラフを選択する方法について説明する．

5.1.4.1 エディタの座標の定義

Ace が定義するエディタの座標はオブジェクトを用いて

以下のように定義される．

row ： 0 から始まる行
column ： 0 から始まる列

また，esprima でプログラムを解析する際に用いられる座標は，オブジェクトを用いて以下のように定義される．

line ： 1 から始まる行
column ： 0 から始まる列

5.1.4.2 カーソル位置情報からチェックポイント ID を取得

チェックポイントを挿入する際，そのチェックポイントが挿入された座標を保存する．座標は __$__.Context.CheckPointTable というオブジェクトにチェックポイントの ID 名をプロパティとして保存される．

表示するグラフを選択する際に用いる関数 FindId (ソースコード 17) は，Ace が定義するエディタの座標を引数として与えると，引数の座標よりも前にあり，最も近いチェックポイントの ID と引数の座標よりも後ろにあり，最も近いチェックポイントの ID を返す．

関数 FindId は全てのチェックポイント ID を 1 から順に調べていき，res オブジェクトを更新していく．そのため，座標が最も近いチェックポイントが複数存在した場合は，チェックポイントの ID の値が小さい方が優先される．

5.1.4.3 Snapshot View Mode の場合

Snapshot View Mode のときのグラフの選択方法を述べる．表示されるグラフは__$__.Context.StoredGraph オブジェクトに保存されており，グラフを選択するには，

- チェックポイントの ID
- ループの ID
- ループの可視化文脈

の情報をプロパティ名として渡す．

関数 FindId を用いて現在のエディタ上のカーソル位置よりも手前に存在し，かつ最もカーソル位置に近いチェックポイントの ID を取得する．この ID の選択方法を用いることで，カーソル位置までプログラムを実行した時点でのグラフを表示することを実現している．チェックポイントはループの ID をちょうど 1 つ持つため，__$__.Context.StoredGraph のプロパティにチェックポイントの ID を渡すことでループの ID を取得することができる (ソースコード 18 の 2 行目)．また，ソースコード 18 の 3 行目でループの可視化文脈を取得している．

5.1.4.4 Summarized View Mode の場合

Summarized View Mode のときのグラフの選択方法を述べる．表示されるグラフは，変換後のプログラムを評価した際に最後に挿入されているチェックポイントで保存されているグラフである．ただし，Summarized View Mode の場合は Probe が行われないため，取得したグラフから透明な頂点及び変数を表す矢印は除かれる．

カーソル位置の命令で起こる変化を，色や辺の形式を変更することで表現する．関数 FindId を用いて現在のエディタ上のカーソル位置の前後で最も近いチェックポイントの ID を取得し，このチェックポイント間で変更されたグラフ構造を調べることで実現する．

まず前提として，2 つのチェックポイントのループの ID が異なっていた場合は，最後のチェックポイントで保存されたグラフを変更せず表示する．

前後のチェックポイントにおけるループの ID が同じであった場合，それら 2 つのチェックポイントで保存された全てのグラフ構造を取得し，同じループ回数のときのグラフを比較し，差分を計算する．

グラフの頂点の識別は，頂点に与えられている ID から判

ソースコード 17: カーソル位置の前後で最も近いチェックポイントの ID を探す関数 FindId

```
1  __$__.Context.FindId = function(pos) {
2      let before, after;
3      let res = {};
4
5      Object.keys(__$__.Context.CheckPointTable).forEach(
           function(key) {
6          let temp = __$__.Context.CheckPointTable[key];
7
8          // the case that temp can become before
9          if (temp.line < pos.row + 1 || temp.line == pos.
               row + 1 && temp.column <= pos.column) {
10             if (!before || before.line < temp.line ||
                   before.line == temp.line && before.
                   column <= temp.column) {
11                 before = temp;
12                 res.beforeId = key;
13             }
14         } else {
15             if (!after || temp.line < after.line ||
                   after.line == temp.line && after.column
                   >= temp.column) {
16                 after = temp;
17                 res.afterId = key;
18             }
19         }
20     });
21
22     return res;
23 };
```

ソースコード 18: 保存されたグラフから描画するグラフを選択する

```
1  var checkPointId = __$__.Context.FindId(__$__.editor.
       getCursorPosition());
2  var loopId = Object.keys(__$__.Context.StoredGraph[
       checkPointId.beforeId])[0];
3  var count = __$__.Context.LoopContext[loopId];
4  var graph = __$__.Context.StoredGraph[checkPointId][
       loopId][count];
```

断する．グラフの辺の識別は，辺の両端の頂点の ID 及び辺のラベルから判断する．そしてグラフの差分を計算し，その差分を表示するグラフに加えて表示する．ここで，表示される差分は次の場合が考えられる．

5.1.4.4.1 頂点が追加された場合

頂点が追加された場合は，その頂点の color プロパティを"orange"に指定する．

5.1.4.4.2 辺が追加された場合

辺が追加された場合は，その辺の color プロパティを"orange"に指定する．また，最後のチェックポイントで保存されたグラフに辺が存在しない場合は，dashes プロパティを true にすることで破線の辺に変更する．

5.1.4.4.3 辺が除かれた場合

辺が除かれた場合は，その辺の color プロパティを"seagreen"に指定する．また，最後のチェックポイントで保存されたグラフに辺が存在しない場合は，同様に dashes プロパティを true に指定する．

5.2 Jump to Construction

本節では Jump to Construction の実現方法について説明する．

Jump to Construction を実現するために，グラフ構造の全ての頂点及び辺が生成されたコード位置を把握しなければならない．ここでいう「コード位置」というのは，

- エディタ上での座標
- 文脈

を表している．そこで，グラフ構造の頂点及び辺が生成されたコード位置を把握するために，変換後のプログラムに挿入されているチェックポイントを利用する．

5.2.1 チェックポイントでの操作

チェックポイントで，保存されたグラフ構造に対して，そのチェックポイントで初めて追加された頂点または辺が存在するかを調べる．そして，初めて追加された頂点及び辺がある場合は，__$__.JumpToConstruction.GraphData にその頂点及び辺の情報と，コード位置を保存する．

ソースコード 19 は，頂点について調べた例である．ソースコード 19 では，変数 flag が初めて追加された頂点かどうかを判定する変数であり，初めて追加された頂点であるときは__$__.JumpToConstruction.GraphData に頂点の ID とコード位置が追加される (ソースコード 19 の 8 行目から 14 行目)．

辺について調べる場合は，5 行目の判定及び 10 行目で用いられている頂点の情報を，辺の両端 (from，to プロパティ) とラベル (label プロパティ) に変更したプログラムを用いる．

5.2.2 vis.js のイベントハンドラ

vis.js で表示されるグラフには，イベントハンドラを定義することができる．イベントハンドラとは，クリックやドラッグといった「イベント」が発生した際に実行される処理のことである．

ソースコード 19: チェックポイントで新しく追加された頂点を調べる

```
1  storedGraph.nodes.forEach(node => {
2      var flag = false;
3
4      __$__.JumpToConstruction.GraphData.nodes.forEach(
           nodeData => {
5          flag = flag || (node.id == nodeData.id);
6      });
7
8      if (!flag)
9          __$__.JumpToConstruction.GraphData.nodes.push({
10             id: node.id,
11             loopId: loopId,
12             count: count,
13             pos: __$__.Context.CheckPointTable[
                   checkPointId]
14         });
15 });
```

ソースコード 20: グラフ構造にイベントハンドラを定義

```
1  __$__.network.on("click", __$__.JumpToConstruction.
       ClickEventFunction);
```

Jump to Construction を実現するため，vis.js で描画されたグラフ構造がクリックされた際に__$__.JumpToConstruction.ClickEventFunction という関数を呼び出すよう設定した (ソースコード 20)．

関数__$__.JumpToConstruction.ClickEventFunction では，次の処理が行われる．まず，クリックされた要素が頂点か辺かそれ以外かを判別する．そして，頂点である場合はクリックされた頂点の ID を，辺である場合はクリックされた辺の両端の頂点の ID 及び辺のラベルを保存する．そして，Snapshot View Mode に切り替えた後，クリックされた要素が初めて追加されたコード位置を__$__.JumpToConstruction.GraphData から取得し，可視化文脈の変更及びエディタのカーソルの移動が行われる．

5.3 工夫点

5.1 節及び 5.2 節では，Kanon のデータ構造ライブ可視化機能を実現する方法を述べた．本節では，Kanon の可視化機能を実装する上で工夫した点を述べる．

5.3.1 グラフの概形の保持

プログラムを編集するたびにグラフを再描画する場合，再描画されるグラフは毎回異なる形のグラフが表示される．しかし，再描画される度に異なる形のグラフが表示さ

れていては，プログラムを編集している途中にデータ構造を観察することは不可能である．そこで我々は，描画時及び頂点移動時にグラフの頂点の座標を保存することで，グラフを再描画する際にグラフの概形を保つよう設計した．

5.3.1.1 vis.js の座標

グラフを表示するために用いている vis.js は，頂点の座標をオプションで加えることが可能である．vis.js の座標は x，y プロパティで定義され，右方向に x 軸，下方向に y 軸が設定されている．

しかし，vis.js でグラフを表示する際，表示される範囲にグラフが収まるようになっているため，vis.js の座標は「相対座標」であると言える．

5.3.1.2 グラフの頂点の座標の保存

再描画の際にグラフの概形を保つため，表示されているグラフの頂点の座標を記録し，再描画時に利用する．

まず初めに，プログラムを評価し終えた時点におけるグラフ構造を描画し，そのグラフの座標及びその頂点の ID を`__$__.StorePositions.oldNetworkNodesData`というオブジェクトに保存する．そして，可視化モードによって表示されるグラフ構造は変化するが，カーソル位置を移動することによってグラフが再描画される場合，再描画されるグラフの頂点の ID と`__$__.StorePositions.oldNetworkNodesData`に保存されている頂点の ID を比較し，一致する ID の頂点の座標を設定して再描画する．

再度プログラムが編集された場合は，既に保存されている座標と比較する．新しい頂点が追加されている場合は，まず既存の頂点の `fixed` プロパティを `true` と設定することで固定する．そして，新しい頂点が他の全ての頂点と反発しながら安定するまで移動し，グラフが安定した状態で全ての頂点の `fixed`，`physics` プロパティを `false` に変更した上で新規の頂点の座標を`__$__.StorePositions.oldNetworkNodesData`に保存する．ここで，`physics` プロパティは，その頂点が反発する対象であるかを指定するオプションである．また，プログラムの編集によって，既存の頂点が取り除かれた場合は，`__$__.StorePositions.oldNetworkNodesData`からも取り除かれる．

Kanon で表示されるグラフは，頂点をドラッグアンドドロップすることで自由に移動することが可能である．これは，描画されている頂点をドラッグアンドドロップによって移動した場合，ドラッグ終了時に再び`__$__.StorePositions.oldNetworkNodesData`に座標を保存することで対応した．

5.3.2 new 式とループの ID の付け方

5.1.2.2.2 段落及び 5.1.2.2.3 段落で述べたように，プログラム中の `new` 式及びループにはそれぞれ固有の ID が与えられる．この固有の ID は，通常は抽象構文木の変換の際に順番につけられる．`new` 式の固有の ID は`"new1"`，`"new2"`と，ループの ID は`"loop1"`，`"loop2"`となる．

ソースコード 21: new 式及びループを 1 つずつ追加する

```
1 new Array();
2 while (false) {}
```

しかし，プログラムの先頭にソースコード 21 の 2 行を追加する場合を考える．この時，`new` 式及びループの固有の ID は抽象構文木の変換の際に順番につけられるため，追加前のプログラムの `new` 式の ID とループの ID は全て 1 ずつ増加する．

`new` 式の ID を用いることでグラフ構造のオブジェクトに ID をつけていた．また，グラフの頂点の座標は`__$__.StorePositions.oldNetworkNodesData`に頂点の ID と共に保存されていたため，グラフの概形が保たれていた．しかし，`new` 式の ID が変更されてしまうと保存されている頂点の座標を描画時に用いることができない．

また，可視化文脈を保存している`__$__.Context.LoopContext` オブジェクトは，ループの ID とその可視化文脈を保存しているため，ループの ID が変更されてしまうと文脈に「ズレ」が生じてしまう．

そこで，`new` 式及びループの ID をつける際に，`new` 式及びループの ID とプログラム上の座標を以下のオブジェクトに保存する．

- `__$__.Context.NewIdPositions` ： `new` 式の ID と座標
- `__$__.Context.LoopIdPositions` ： ループの ID と座標

そしてプログラムの編集が行われた際，その編集によってプログラムが移動した部分に存在する `new` 式及びループの座標を更新する．例えば，ソースコード 21 の 2 行をプログラムの先頭に追加した場合は，`__$__.Context.NewIdPositions` 及び`__$__.Context.LoopIdPositions` に保存されている全ての座標の `line` プロパティの値が 2 増える．

そして抽象構文木の変換において `new` 式及びループの ID をつけるとき，その `new` 式またはループの座標と保存されている座標を比較し，一致した場合はその一致した座標の ID を与える．このようにすることで，プログラムの編集によって `new` 式とループの ID は変更されなくなる．

5.3.3 Probe での変数のスコープ管理

JavaScript の変数宣言には `var`，`let`，`const` の 3 種類の宣言方法が存在する．`var` を用いて宣言された変数のスコープは関数内に制限され，`let`，`const` を用いて宣言された変数のスコープはブロック内に制限される．

Probe の対象の変数がグラフ上に表示されるのは，カーソルが Probe 対象の変数の有効範囲内にあるときのみであ

ソースコード 22: 変数のスコープを管理するスタック及びそのフレームのコンストラクタ

```
1  __$__.Probe.FunctionFlame = function() {
2      this.next = null;
3      this.prev = null;
4      this.env = {};
5  };
6
7  __$__.Probe.BlockFlame = function() {
8      this.next = null;
9      this.prev = null;
10     this.env = {};
11 };
12
13 __$__.Probe.StackEnv = function() {
14     this.head = new __$__.Probe.FunctionFlame();
15     this.tail = new __$__.Probe.BlockFlame();
16     this.head.next = this.tail;
17     this.tail.prev = this.head;
18 };
```

ソースコード 23: 変数が Probe の対象であるかどうかの情報をスタックに追加する関数 addVariable

```
1  __$__.Probe.StackEnv.prototype.addVariable = function(
       variable, kind, visualize) {
2      var current = this.tail;
3
4      if (kind === "var") {
5          while (!(current instanceof __$__.Probe.
               FunctionFlame))
6              current = current.prev;
7
8          current.env[variable] = visualize;
9      } else {
10         while (!(current instanceof __$__.Probe.
               BlockFlame))
11             current = current.prev;
12
13         current.env[variable] = visualize;
14     }
15 };
```

る．そこで，カーソル位置において有効ではない変数が表示されないよう，変数のスコープを独自で管理する必要がある．以下では，変数のスコープ管理をどのように実現したか述べる．

5.3.3.1 スタックの定義

変数のスコープを管理するため，スタックを用意する (ソースコード 22)．スタックに格納されるデータは FunctionFlame または BlockFlame である (ソースコード 22 の 1 行目, 7 行目)．また，スタックには既に FunctionFlame 及び BlockFlame が 1 つずつ格納されている (ソースコード 22 の 14, 15 行目)．

5.3.3.2 スタックを用いた変数のスコープ管理

抽象構文木を走査すると同時にソースコード 22 のコンストラクタで定義されるスタックを変更することで，変数のスコープ管理を実現する．

スタックに格納される 2 種類のフレームについて説明する．FunctionFlame は抽象構文木の走査の際に以下のノードに入った場合にスタックに追加される．

- FunctionExpression
- ArrowFunctionExpression
- FunctionDeclaration

そして，これらのノードの子の走査が終了した場合はスタックの先頭の FunctionFlame はスタックから取り除かれる．BlockFlame は抽象構文木の走査の際に BlockStatement のノードに入った場合にスタックに追加され，BlockStatement のノードの子の走査が終了した場合はスタックの先頭の BlockFlame はスタックから取り除かれる．

各フレームには env プロパティが定義されている．env プロパティのオブジェクトは変数が Probe の対象であるかを格納するオブジェクトである．env プロパティが参照するオブジェクトには，プロパティとして変数名，値として Probe の対象かどうかの真偽値を指定する．そして，値が true である場合はその変数が Probe の対象であることを表し，false である場合は対象でないことを表す．

抽象構文木を走査し，関数またはブロックのノードが現れた場合にスタックにフレームが追加されるとき，同時にブロック内に存在する変数宣言を取り出し，その変数宣言の

1. 宣言される変数名
2. 変数宣言の種類 (var, let, const)

を確認する．そしてスコープによる変数の初期化のため，上記の変数は Probe の対象でないという情報をスタックに追加する．変数が Probe の対象であるかどうかの情報は，addVariable 関数を用いて追加される (ソースコード 23)．第 1 仮引数 variable は変数名，第 2 仮引数 kind は変数宣言の種類，第 3 仮引数 visualize は Probe の対象であるかを表す．ここでは，初期化のために addVariable の第 3 引数は false とする．

また，走査で変数宣言のノードの処理をする際，

1. 宣言される変数名
2. 変数宣言の種類
3. $の有無

を確認し，addVariable 関数を用いて変数が Probe の対象であるかどうかの情報をスタックに追加する．

5.3.3.3 Probe の対象の変数

チェックポイントの第 6 引数には，Probe の対象となっている変数の変数名及びその値を格納したオブジェクトを渡す．そこで，スタックのフレームを tail から head まで確認し，そのチェックポイントで Probe の対象となっ

ソースコード 24: 多重ループの例

```
1  var f = function() {
2      // (1)
3      for (var i = 0; i < 5; i++) {
4          // (2)
5      }
6  };
7
8  f();
9  f();
10 f();
```

ている変数を把握する．そして対象の変数はチェックポイントの第 6 引数のオブジェクトに {変数名: eval(変数名)} という形で追加される．

5.3.4 文脈が変更された時の修正

可視化文脈は，__$__.Context.LoopContext オブジェクトにループの ID 及び周回数という形で保存されている．しかし，可視化文脈を周回数で保存しているため，多重ループの場合に文脈に「ズレ」が生じる可能性がある．

ここでいう「ズレ」について，ソースコード 24 の例を考える．プログラム中の関数 f は 3 回呼び出されており，1 回の関数呼び出しで for 文は 5 周実行されるため，(1) の部分は 3 回，(2) の部分は 15 回処理される．つまり，関数 f の可視化文脈は 1 から 3 のいずれかであり，for 文の可視化文脈は 1 から 15 のいずれかである．可視化文脈の初期値は 1 であるため，ここでは両方の可視化文脈は 1 であるとする．このとき，関数 f の可視化文脈を 2 に変更した場合，関数 f の可視化文脈が 2 であるのにも関わらず for 文の可視化文脈は 1 であり，これは直感的でないと考える．

このような「ズレ」の対処として，プログラム実行時の時間軸を管理する__time_counter という変数を用いる．変数__time_counter は変換されたプログラムの先頭で宣言され (ソースコード 4 の 6 行目)，チェックポイントを通過する度に 1 ずつ増加していく．そして，__$__.Context.StartEndInLoop というオブジェクトにループが実行された時間関係を保存する (ソースコード 6 の 24 行目)．この__$__.Context.StartEndInLoop に保存された時間関係を利用して，可視化文脈が変更される度に可視化文脈の「ズレ」を修正する．

6. 関連研究

YinYang [10] は独自の環境下でライブプログラミングを可能にする言語である．YinYang には，プログラム中の変数や式の値を表示する Probing という機能や，prn 関数の引数として渡された式を評価した結果のトレースを表示する Tracing と呼ばれる機能が備わっている．Kanon の Snapshot View Mode や Jump to Construction，Probe はこの YinYang 言語の影響を強く受けている．YinYang 言語は指定された文脈で Probing を表示するが，Kanon の Summarized View Mode のように全ての変化を同時に表示することはない．また，YinYang は数値計算には便利であるが，データ構造の扱いには適していない．

SuperGlue [8] は eclipse のプラグインとして作られた環境下で機能するライブプログラミング言語であり，SuperGlue は signal 及び動的継承を用いて実現されるオブジェクト指向なデータフロープログラミングを実現する．しかし，SuperGlue ではプログラム内で生成されたデータ構造に関して描画しない．

Khan Academy の Live Editor [15] は JavaScript とそのライブラリである Processing.js のライブプログラミング環境であり，数値をドラッグアンドドロップで変更するスライダなどのプログラマを支援する機能が含まれている．また，Kanon は Live Editor のデザインを参考に作成されている．しかし，Live Editor も SuperGlue 同様にデータ構造に関して描画しない．

Theseus [7] は Brackets という IDE の拡張で，実行時の振る舞いを可視化する JavaScript 用のエディタである．ここで言う「実行時の振る舞い」とは，コールバックの無名関数を含む全ての関数が呼び出された回数やその返り値のことを表し，関数呼び出しはイベントハンドラのような「動的」な呼び出しにも対応している．また，プログラマが指定した関数のログを実行された順に表示する機能も備えている．Theseus は実行時の情報をライブに可視化するという点では Kanon と似ているが，Theseus にはデータ構造の可視化であったり，Snapshot View Mode のように実行途中の状態を確認できない．また，Theseus は完成したプログラムの挙動を確認することを目的としている．そのため，記述したプログラムが保存された時に実行するので，Kanon のように現在編集している部分がどのように構造に影響しているのかを確認しながらプログラムを記述することができない．

そのプログラムを全て読むのは，理解する人にとって負担が大きい．そこで，プログラムに記述されているデータ構造を把握する手段として，データ構造を可視化するという研究がなされている．

ghc-vis [3] は Haskell のデータ構造可視化ツールであり，Haskell のプログラムのデータ構造を図表現として可視化する．また，Haskell には lazy evaluation や sharing といった特徴があり，ghc-vis を用いることでこれらの挙動を図表現として見ることができる．しかし，ghc-vis は Haskell の対話環境 GHCi 上で用いるため，Kanon のようにプログラムの時間軸を戻したグラフを表示する場合には再定義をしなければならない．また，ghc-vis で表示されるグラフの各ノードは Kanon のように移動することができない．

Python Tutor [4] は Web ブラウザ上で動作する Python のためのプログラム可視化ツールで，記述したプログラムを

実行した際のデータ構造を図表現として表示する．Python Tutor は実行の 1 ステップごとにグラフを生成するため，ユーザが指定したステップのグラフを表示することが可能である．また，現在では Python 以外に Java，JavaScript，TypeScript，Ruby，C，C++ の言語にも対応しており，更に Python 及び JavaScript にはライブプログラミング環境も備わっている．しかし，Python Tutor はプログラムを入力してからデータ構造可視化まで 1 秒程度の時間がかかる．これは，ライブプログラミングにとって重要な「即時なフィードバック」に反している．また，Python Tutor で可視化されるグラフは Kanon のように移動することができず，表示される図表現の見やすさも十分な表現ではない．

Igor [2] は巻き戻しステップ実行が可能なデバッガである．巻き戻しステップ実行はプログラムを理解する上で非常に強力な機能である．しかし，通常プログラムの編集とデバッガのステップ実行は異なるプロセスである．実際，ブログラムをステップ実行し，バグを発見した場合はステップ実行を一旦中止する．そしてプログラムを編集し，編集が完了したら再びデバッグを行う．一方で Kanon の Snapshot View Mode はプログラムをカーソル位置まで実行した時点のグラフ構造を表示しているため，カーソルの移動によって巻き戻しステップ実行は可能である．また，プログラムを編集しながらステップを進めることも可能である．

7. 結論と今後の課題

7.1 結論

本研究では，データ構造のためのライブプログラミング環境 Kanon を提案した．具体的には，データ構造の定義や操作を行うプログラムを記述する際にプログラマの負担を軽減する機能を設計，実現した．

カーソル位置の命令の影響を表示する二種の方法として，Snapshot View Mode と Summarized View Mode を提案した．Snapshot View Mode ではユーザが指定した位置までプログラムを実行した時点でのグラフを表示する．また，Summarized View Mode ではある命令の変化を「一斉に」表示することで，その命令の変化を確認する際に各文脈の命令の変化を 1 つずつ確認する手間を省くことができた．変数を表す矢印をグラフ上に表示する Probe によって，プログラム中の変数が参照しているオブジェクトを把握できるようにした．また，Kanon を用いる上での機能として，文脈を変更する手段及び生成されたコード位置への移動を可能にする Jump to Construction や，表示されるグラフの見やすさを向上するため，プログラマがグラフの頂点を自由に移動できる機能など Kanon をより使いやすくする機能を提案した．

Kanon は 72 行の HTML と 28 行の CSS，約 1,900 行の JavaScript で構成されており，外部ライブラリとして Ace, esprima，escodegen，vis.js を用いて実装した．そして，プログラム変換をし，プログラムの各文の前後にチェックポイントを挿入することで，Snapshot View Mode や Summarized View Mode でカーソル位置に依存したグラフの表示を実現した．また，描画時及び頂点移動時にグラフの頂点の座標を保存することで，グラフの図表現を再描画する際にグラフの概形を保つよう設計し，実現した．

また，双方向連結リストのプログラムを例題とした事例研究を行い，Kanon がプログラマに貢献する場面を考察した．

7.2 今後の課題

本節では，本研究の今後の課題を述べる．

7.2.1 Snapshot View Mode の可視化文脈

7.2.1.1 現在の可視化文脈の表示

Kanon には Snapshot View Mode における可視化文脈が存在するが，プログラマがその可視化文脈を確認する手段が「可視化文脈を確認したいループに Snapshot View Mode のときにカーソルを合わせ，表示されるグラフの状態を見る」しか存在しない．そこで，Kanon の課題の一つとして，プログラマが可視化文脈を一目で確認できるように各ループの付近に可視化文脈を表示できるようにする必要がある．

表示される可視化文脈は，現在指定されている可視化文脈を表示する方法以外に，ループ変数の値を表示する方法が考えられる．ただし，ここでいうループ変数とは，ループの周回とともに変化する変数のことを表す．

7.2.1.2 可視化文脈の変更方法

現在，可視化文脈を変更する方法が Jump to Construction しか存在しない．そのため，可視化文脈を変更する際には，ループ内で生成されているオブジェクトまたは参照関係をクリックしなければならない．しかし，ループ内でオブジェクト及び参照関係が生成されない場合はそのループの可視化文脈を変更できないという問題が生じてしまう．

可視化文脈の変更方法として，以下の案が考えられる．

- キー入力による可視化文脈移動
- スライダでの可視化文脈移動

キー入力ついては，コマンドを入力するとカーソル位置のループの可視化文脈を変更するというものである．しかし，キー入力を用いた可視化文脈の変更を行う場合，1 つのコマンドで可視化文脈を 1 つ増やすことにすると，非常に周回数が多いループの可視化文脈を移動する際にユーザの負担が大きくなってしまう．そのため，コマンドでの可視化文脈移動以外の手段も考える必要がある．

スライダを用いた可視化文脈の移動は，ドラッグアンドドロップで可視化文脈を変更するものである．Khan Academy の Live Editor [15] は数値の変更にスライダを採用している (**図 16**).

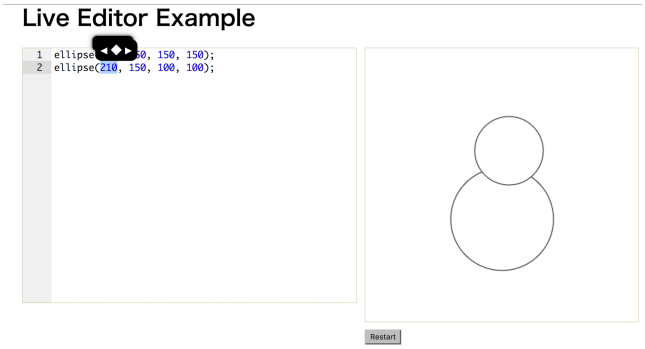

図 16: Live Editor [15] のスライダ
Fig. 16 A slider in Live Editor [15].

7.2.2 多数のオブジェクトを生成するプログラムへの対応

現在 Kanon で表示されるグラフは頂点及び辺が存在し，それぞれにオブジェクト名やプロパティ名が記されている．また，表示されるグラフの頂点は移動でき，再描画の際にグラフの大きさが出力画面の枠内に収まるようグラフの大きさが自動的に変更される．

しかし，プログラムの規模が大きくなるにつれて表示されるグラフの頂点または辺の数が増え，グラフ表示にかかる時間や Kanon の処理速度に影響を及ぼす．また，数多くの頂点が出力画面に表示されるとグラフの頂点 1 つあたりの大きさが小さくなってしまう．

そのため，大規模なプログラム開発にも対応できるよう，表示するグラフをユーザが選択した一部分のみの表示に変更することが本研究の課題の 1 つである．

7.2.3 その他

以上で述べた改良すべき点の他に，現在の Kanon では対応していない非同期処理の対応や，表示するグラフの種類をユーザが選択できるようにすることが今後の課題としてあげられる．

参考文献

[1] Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Sean McDirmid, Michal Moskal, Nikolai Tillmann, and Jun Kato. It's Alive! Continuous Feedback in UI Programming. *SIGPLAN Not.*, 48(6):95–104, June 2013.

[2] Stuart I. Feldman and Channing B. Brown. IGOR: A System for Program Debugging via Reversible Execution. *SIGPLAN Not.*, 24(1):112–123, November 1988.

[3] Dennis Felsing. Visualization of Lazy Evaluation and Sharing. Bachelor's thesis, Karlsruhe Institute of Technology, Germany, September 2012.

[4] Philip J. Guo. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[5] Chirstopher Micahel Hancock. *Real-Time Programming and the Big Ideas of Computational Literacy*. PhD thesis, Massachusetts Institute of Technology, 2003.

[6] Tomoki Imai, Hidehiko Masuhara, and Tomoyuki Aotani. Making Live Programming Practical by Bridging the Gap between Trial-and-Error Development and Unit Testing. In *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pages 11–12, New York, NY, USA, 2015. ACM.

[7] Tom Lieber, Joel R. Brandt, and Rob C. Miller. Addressing Misconceptions About Code with Always-On Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2481–2490, New York, NY, USA, 2014. ACM.

[8] Sean McDirmid. Living it up with a Live Programming Language. *SIGPLAN Not.*, 42(10):623–638, October 2007.

[9] Sean McDirmid. sqrt - YouTube. `https://www.youtube.com/watch?v=01Xyoh-G6DE`, April 2013. Accessed 2016-12-26.

[10] Sean McDirmid. Usable Live Programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 53–62, New York, NY, USA, 2013. ACM.

[11] Apple Inc. Swift - Apple Developer. `https://developer.apple.com/swift/`. Accessed 2017-1-23.

[12] Applicative. Haskell for Mac IDE - Learn Functional Programming with Haskell. `http://haskellformac.com/`. Accessed 2017-1-25.

[13] Jason Park. Algorithm Visualizer. `http://algo-visualizer.jasonpark.me/`. Accessed 2017-1-25.

[14] Kodawa Inc. Light Table. `http://lighttable.com/`. Accessed 2017-1-25.

[15] John Resig. Redefining the Introduction to Computer Science. `http://ejohn.org/blog/introducing-khan-cs/`. Accessed 2016-12-23.

[16] Steven L. Tanimoto. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming*, LIVE '13, pages 31–34, Piscataway, NJ, USA, 2013. IEEE Press.

[17] W. Teitelman and L. Masinter. The Interlisp Programming Environment. *Computer*, 14(4):25–33, April 1981.