令和4年度 修士論文

実行履歴を用いた差分実行型ライブプログラミング環境の実現

東京工業大学 情報理工学院
数理・計算科学系 数理・計算科学コース

学籍番号 21M30212

高橋 修祐

takahashi.s.bq@m.titech.ac.jp

指導教員

増原 英彦 教授

令和5年1月30日

概要

ライブプログラミング環境はプログラムが編集されると即座に実行をし，結果を表示する。この機能により，プログラマはプログラムが意図通りに振る舞うことを指揮や文といった細かい単位で確認できる。例えば，データ構造ライブプログラミング環境である Kanon は，与えられたプログラムの実行中に生成されるオブジェクトの参照関係（オブジェクトグラフ）を可視化し表示する。

ライブプログラミングではプログラムに微小な変更を加えて最実行することを繰り返すため，直前のプログラムの実行と変更後のそれは大部分が同じ計算をすると考えられる。一方で，既存のライブプログラミング環境は編集が発生するたびにプログラムを先頭から実行し直すため，同じ計算が何度も繰り返されるという問題が有る。

我々は，Kanon のような環境がもともと可視化のために実行履歴を記録していること，及び変更前後の実行の大部分が同じ計算であることに着目し，プログラム編集によって再計算が真に必要となった箇所のみを再実行し実行時情報を効率的に収集する**差分実行方式**を提案する。具体的には，まず編集前の実行において，各式がどの値を用いて計算されたかを記録しておく。この情報とプログラムの差分を用いることにより，ある式の実行内容が編集によって変化している可能性があるかをその式の実行前に判定する。変化の可能性がない場合には，その式の実行で発生するべき実行時情報を前回実行時の履歴から再利用し，編集によって影響を受けている可能性がある部分については，その部分に限って再実行を行い新しい実行時情報を収集する。

本論文ではこの差分実行方式を実現するために必要な実行時情報を定義し，差分実行方式を用いた実行時情報収集手法を示した。また，実際の実現における効率的な再実行方法や履歴構築方法を示した。更にこの方法に従い，Graal/Truffle 上で実現された小規模言語をターゲットとする Kanon 仮想機械を構築した。評価として，本手法を導入した処理系と通常の処理系とで編集前後の両プログラムを実行した。その結果，編集による影響を受ける箇所が小さい場合には実行時情報の収集が効率よく行えていたこと，特に編集差分が小さいときに効果を強く発揮したことを確認した。

謝辞

本研究について，まず増原英彦教授と叢悠悠助教のお二方に御礼申し上げます。増原先生には，研究室に配属されてから本日に至るまで，研究の方向性や具体的なアイディアについての議論，実現の詳細，適切な著述や発表の方法など，研究活動全般において手厚くご指導を賜りました。また叢先生にも，主に研究発表に関して非常に多くの有益なご助言をいただきました。これら全てのご指導は，今後の私の研究活動，あるいは人生全般において，私の核を成すことは言うまでもありません。

続いて，増原研究室に所属する学生の皆様にも感謝を表します。所謂コロナ禍において，研究室の皆様と交流する機会は多くはありませんでしたが，それでもセミナ発表などにおいて，有益でまた鋭い質問によって私に示唆を与えていただいたことについて忘れることはないでしょう。特に，論文執筆や発表について，具体的でかつ極めて有意義な数多くのコメントを頂いた田辺裕大氏には厚く御礼申し上げます。

そして，増原研究室の一員であり，また共同研究者でもある伊澤侑祐氏には感謝してもしきれません。伊澤氏には，研究を開始してから現在までの間，未熟な私を一研究者として対等に，しかし非常に親身になって，私の研究に関するあらゆることから今後の研究者としての在り方に至るまでをご教授いただきました。この 3 年間に得られた成果は間違いなく伊澤氏の薫陶の賜物であり，伊澤氏と共に得た経験は礎となり今後においても私を支え続けることでしょう。

また学士課程に進学して以降，普段は協力相手として，ときには競争相手となり私を励ましてくれた同期の友人たちにも御礼申し上げます。

最後に，学業を修めるにあたり常に支援と応援をしてくれた両親に深く感謝申し上げ，謝辞とします。

目 次

図 目 次

表 目 次

アルゴリズム目次

第1章　はじめに

ライブプログラミング環境は，編集されたプログラムを即座に再実行し，その実行内容を利用者に表示する。この機能により，利用者はプログラムに対する小さな編集がプログラム実行に及ぼす影響を直ちに視認できる [16]。既存のライブプログラミング環境には，ブロックを用いたプログラミング言語とその環境 Scratch [10] や，カメラ入力を用いたインタラクティブな開発を支援する DejaVu [7]，音楽プログラミング環境 [1]，データ構造の可視化に特化した Kanon [13] などが挙げられる。

我々は，このうち Kanon をはじめとするデータ構造に特化したライブプログラミング環境の即応性を高めることを目標とする。データ構造に特化した環境は，データ構造の観点からフィードバックを行うことで，プログラムが意図したとおりに動作しているかを環境利用者が確認できるようにする。例として Kanon (図 1.1) は，プログラム実行中のオブジェクトの参照関係（オブジェクトグラフ）をノードリンク図として可視化し表示する。環境に入力されたプログラムが編集されると，環境は即座にそのプログラムを再実行してオブジェクトグラフを収集し描画する。

ライブプログラミング環境において即応性に関わる要素は複数あるが，本論文ではプログラム実行中にフィードバックに必要な情報を効率良く収集することに焦点をあてる。データ構造特化の環境は実行中にオブジェクトグラフを収集するが，編集前に得られた情報とプログラムの編集差分を活用することで効率的に収集ができる。例えば既存実現手法の 1 つである Omniscient Debugger [8] は，プログラム実行中のすべてのイベントを記録する。また，この記録を用いることによりプログラム実行中のある時点における環境を再構築することも可能である。これにより，プログラムの実行内容が編集によって初めて変化する箇所までの実行を省略し，その時点の環境を再現して再実行を開始することが可能になり，したがって実行時情報の収集を効率的に行える。

我々は，前回の実行時履歴とプログラム編集の差分をさらに活用することにより，プログラム編集によって再計算が真に必要となった箇所のみを再実行し実行時情報を効率的に収集する**差分実行方式**を提案する。具体的には，まず編集前の実行において，各式がどの値を用いて計算されたかを記録しておく。この情報とプログラムの差分を用いることにより，ある式

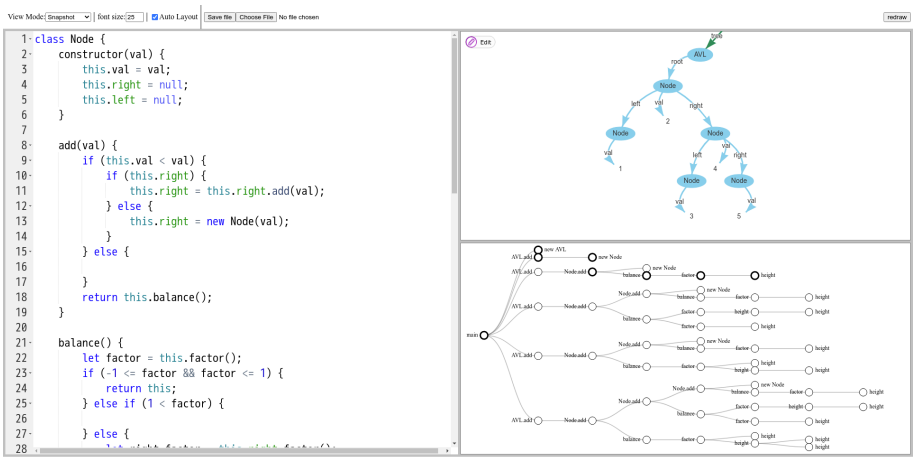

図 1.1: Kanon の画面。左はエディタ，右上はオブジェクトグラフに基づくノードリンク図，右下は実行時の関数呼び出しや反復構造を表す木。

の実行内容が編集によって変化している可能性があるかをその式の実行前に判定する。変化の可能性がない場合には，その式の実行で発生するべき実行時情報を前回実行時の履歴から再利用し，編集によって影響を受けている可能性がある部分については，その部分に限って再実行を行い新しい実行時情報を収集する。

本論文では，提案手法である差分実行方式について詳細なプログラム実行アルゴリズムを示すとともに，効率的な実現方法を詳細に述べる。また，小規模な言語である SimpleLanguage 言語定義に提案手法を導入し，効率的な Kanon ランタイムを構築する。更に差分実行方式を導入した処理系が，そうでない処理系と比べて，プログラム編集後の履歴収集をどれほど効率的に行えるのか評価を行う。

本論文の構成は次のとおりである。第 2 章でライブプログラミング環境 Kanon の詳細な実現方式を述べ，また提案手法の実現対象である言語 SimpleLanguage について説明する。第 3 章では既存環境の実現方法を取り上げ，それらがプログラムの差分を用いてどのように実行時情報を収集できるかを検討し，その後本論文の提案手法である差分実行方式の概略を述べる。第 4 章では，差分実行方式について形式的な観点からどのようにプログラムを実行し実行時情報を収集するかを述べ，また第 5 章では，実際に SimpleLanguage 処理系に対して差分実行方式を導入した際の詳細な実現方式を説明する。第 6 章では，差分実行方式が導入された SimpleLanguage 処理系が，プログラムの差分情報と前回実行時の履歴を用いることによりどれほど効率的に実行時情報を収集するかについて，差

分実行方式を導入していない処理系と性能を比較する。第 7 章では関連研究を挙げ，第 8 章でまとめと今後の展望を述べる。

第2章　背景

本論文は，データ構造ライブプログラミング環境 Kanon に差分実行方式を導入し，一部のプログラムの実行を効率化することを目指す。また，実現例として Graal/Truffle 上で実現された言語 SimpleLanguage に対して差分実行方式を導入する。したがって本章では Kanon の基本的な動作と既存の実現方式を述べ，また Graal/Truffle と SimpleLanguage について簡単に紹介する。

2.1　Kanon

Kanon は，データ構造の可視化に特化した JavaScript 向けライブプログラミング環境である [13]。Kanon はエディタ（図 1.1 左側）中のプログラムが変更されるたびにプログラムを再実行する。続いて，プログラム実行がエディタカーソルを通過した時のオブジェクトグラフをノードリンク図として描画する（図 1.1 右上）。開発者は表示された図をもとに，プログラムが意図したとおりに実現されているかを確認しながらプログラムの開発を進めることが可能となる。

また Kanon では単にオブジェクトの参照関係を図示するだけでなく，生成されたノードリンク図を拡大・縮小したり，ノードの位置を変えることができる。さらに，ソースコードの変更によって図の再描画が行われる際に，直前に描画されたレイアウトを維持しようとする。

2.1.1　Kanon の実行手順

Kanon エディタでプログラムが入力されてから，ノードリンク図を描画するまでの実行フローを 2.1 に示す。

1. はじめに，プログラム実行中の情報を取得するために，プログラムを変換する。具体的には，入力されたプログラム全体に対して構文解析を行い，プログラム中のすべての文の前後にチェックポイントを挿入する。

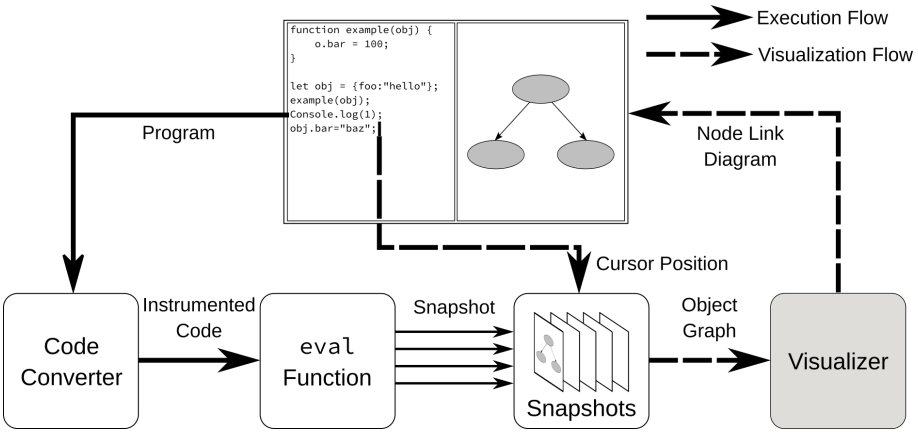

図 2.1: Kanon の実行手順

2. 次に，変換したプログラムを `eval` 関数を用いて実行する。挿入されたチェックポイントが実行されると，第 2.1.2 節に示す情報が，次で示す手順で収集される。

   (a) **オブジェクトの参照関係**: オブジェクトの参照関係をすべて記録する。具体的には，その時点において参照可能なローカル変数及びグローバル変数をすべて取得し，存在する全オブジェクトを推移的に記録する。

   (b) **オブジェクト生成・操作に対応するソースコード上の位置**: チェックポイントのソースコード上の位置を記録する。

   (c) **オブジェクト生成時のコールスタック**: まずチェックポイントが実行された際のコールスタックを取得する。次に 2a で取得したオブジェクトの参照関係の差分をとることによって，新しく生成されたオブジェクトとコールスタックを紐付ける。

3. カーソルの位置を取得し，そこから最も近いチェックポイントで得られた情報を選択する。

4. 選択した情報をもとに，ノードリンク図を生成して表示する。

エディタ上のカーソルの位置が移動した際には，最後に編集されたプログラムから得られた情報に対して，手順 3，4 を再度実行し（2.1 の破線に対応），カーソルの位置に対して適当なノードリンク図を生成する。プログラムが編集された際には，手順 1，2 を再度実行し（2.1 の実線に対応），その後手順 3，4 を実行する。

```
1 function example() {
2     let object = {f: 1}; ▲
3 }
4
5 let x = 1;
6 example();
```

図 2.2: 複数の変数やオブジェクトを持つ JavaScript プログラムの例

2.1.2　Kanon が収集する実行時情報

Kanon は入力プログラムの各ステップが実行されるたびにチェックポイント処理を実行し，その時点における実行時情報を収集する。この情報はオブジェクトグラフの生成に用いられる。Kanon がオブジェクトグラフ生成のために必要とする情報は第 2.1.1 に示した次の 3 つである: (i) オブジェクトの参照関係, (ii) オブジェクト生成・操作に対応するソースコード上の位置, (iii) オブジェクト生成時のコールスタック。以下，この 3 つの情報についてそれぞれ説明する。

(i) オブジェクトの参照関係

各オブジェクトの各フィールドがどのオブジェクトを参照しているのかという情報を，オブジェクトの参照関係と呼ぶ。プログラム実行中におけるある時点のオブジェクトの参照関係は，その時点における次の 3 つの値をすべて取得することによって生成できる。

- ローカル変数の値
- グローバル変数の値
- オブジェクトや配列の各フィールドの値

図 2.2 の 2 行目の文が実行された後（ソースコード中 ▲ の地点）を例に説明するここで，2 行目右辺で生成されたオブジェクトを `A` と表記する。このとき，ローカル変数 `object` の値は `A` であり，グローバル変数 `x` の値は数値 1 であり，オブジェクト `A` のフィールド `f` には数値 1 が保持されている。

(ii) オブジェクト生成・操作に対応するソースコード上の位置

チェックポイント処理が行われた際に，その処理がソースコード上のどの記述に起因するものなのかを記録する。これによりエディタ上のある 1

```
1 function example() {
2     return {f: 1};   ▲
3 }
4                      // 関数呼出ID
5 let x = example();   // example1
6 let y = example();   // example2
```

図 2.3: オブジェクト生成時の関数呼出文脈が異なることを示す JavaScript プログラム例

点を指したときに，その時点におけるオブジェクトグラフを特定し表示することができる。

(iii) オブジェクト生成時のコールスタック

Kanon はオブジェクトグラフを可視化する際に，メンタルマップの保存を試みる [13]。メンタルマップとは，グラフや絵など，出力された視覚的情報を見たときのプログラマの脳内表現である。また，メンタルマップ保存 [2] とは，出力された視覚的表現を更新する際に，それを一から描画し直すのではなく，なるべく更新前の形を保とうとする働きである。Kanon は，プログラマがプログラムを編集したとき，編集前のグラフと編集後のグラフのレイアウト上の差分をなるべく小さくすることによって，変更箇所の影響をイメージしやすくさせる。

Kanon におけるメンタルマップ保存機能を実現するに当たってすべきことは，前回表示したときのノードの位置などの情報を保存することと，プログラムを編集する前後でのオブジェクトを対応付けることの 2 つである。前者は本論文では取り扱わない。後者におけるオブジェクトの対応付けを行うために，Kanon はオブジェクト生成が行われるソースコード上の位置だけでなく，関数呼び出しごとにユニークな ID を振る。生成オブジェクトとソースコード上の位置と関数呼出 ID のペア (これを関数呼出文脈と呼ぶ) を**コールスタック**に記録することで，あるオブジェクト生成がどの関数呼び出しによって行われたかをことができる。これにより，同じ関数が違う文脈で呼ばれた場合でもオブジェクトを区別して描画できる。さらに，プログラムの編集が行われた後にコールスタックを 1 から作り直すのではなく，記録済みのコールスタックを基に関数呼出文脈の記録を再開することで，前回のノードの位置を維持することが可能になる。

コールスタックについて図 2.3 を例に説明する[1]。グローバル変数 `x` と

[1]本論文は関数呼出文脈をコールスタックとして記録することに焦点を当てているため，編集が行われた前後のコールスタックの対応付けの説明は省略する。

5 行目　　6 行目

図 2.4: 図 2.3 の各関数呼出におけるコールスタック

y の値はいずれも 2 行目を実行した際に生成されたオブジェクト ▲ である。x の値が生成された際のコールスタックには 5 行目で関数 example が呼び出されたという情報 (▲: (L5, example1)) が記録される。一方，y の値が生成された際のコールスタックには 6 行目で関数 example が呼び出されたという情報（▲: (L2, example2)）が記録される。6 行目時点でコールスタックは図 2.4 の右側のようになる。このコールスタックを用いて各関数呼び出しによって生成されたオブジェクトを区別することができる。

2.1.3　効率的な実行時情報の収集

Kanon ではスナップショットを用いることによってオブジェクトグラフを収集していたが，オブジェクトグラフの差分，すなわちオブジェクトの更新履歴をもとにオブジェクトグラフを構築することもできる [15]。スナップショットによる方法では，表示されることのないものも含め全てのオブジェクトグラフを生成していた。一方更新履歴をもとにした方法では，表示することが分かって初めてオブジェクトグラフを生成する。この手法では，オブジェクトグラフを表示する際に若干の時間がかかるようになるものの，実行時間を大幅に短縮することが示されている。本論文においてはこの手法に従いオブジェクトグラフ可視化のための履歴情報を収集するため，本手法による実行時情報収集方法とオブジェクトグラフ構築方法を簡単に述べることにする。

実行時に収集する情報

本手法ではまず仮想的な時刻を導入し，各式（もしくは適切な構文上の単位）がいつ評価されたかを時刻によって記録する。この時刻は式を評価するたびにインクリメントされるものである。例えば，図 2.5 に示されたプログラムを考える。ここで単位時刻に対応する構文上の単位は文とする。このプログラム実行時の各文の実行時刻は表 2.1 に示したようになる。

次に，以下の情報を時刻と併せて取得する。

```
1 function example(o) {
2     obj.bar = 100;          // 時刻 t = 2
3 }
4
5 let obj = { foo: "hello" }; // 時刻 t = 1
6 example(obj);               //      t = 3
7 Console.log(1);             //      t = 4
8 obj.bar = "baz";            //      t = 5
9
```

図 2.5: JavaScript プログラムの例

行	時刻の配列
2	[2]
5	[1]
6	[3]
7	[4]
8	[5]

表 2.1: ソースコード上の位置から時刻への対応表

1. 変数への代入
2. オブジェクトのフィールドの初期化・更新

これらの情報は具体的には次のとおりである。変数への代入については，代入された値とその変数名のほか，どのコールスタックにおいて呼び出された関数なのかを取得する。オブジェクトのフィールドに関しては，どのオブジェクトのどのフィールドにどの値が代入されたかを取得する。ここでオブジェクトを判別するために，各オブジェクトには適切な識別子が割り振られているとする。グローバル変数への代入については，グローバル変数全てをフィールドに持つような仮想的なオブジェクト（グローバル変数のルートオブジェクト）の存在を仮定し，グローバル変数のルートオブジェクトの，グローバル変数の名前を識別子とするフィールドに更新が発生したと取り扱う。

次にここで得られた情報を，オブジェクト・フィールドごと（変数の場合はそれぞれコールスタックと変数ごと）の履歴になるように変換する。再度図 2.5 に示したプログラムを例にとることにする。このとき各オブジェクトとフィールドの値の履歴は図 2.6 に示したようになる。

また，オブジェクトが生成されるたびに生成時のコールスタックも記録する。このとき記録する情報は，生成時のコールスタックと生成されたオ

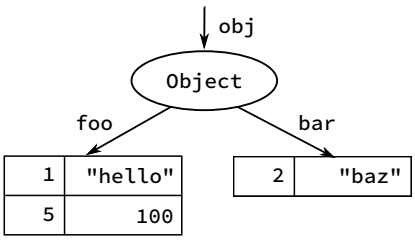

図 2.6: 各オブジェクト・フィールドの値についての履歴

ブジェクトの識別子の 2 つである。

さらに，各関数の呼び出しがいつ始まりいつ終了したかについても記録しておく。

オブジェクトグラフ構築方法

オブジェクトグラフの構築は時刻を指定することによって行う。Kanon のようなエディタ上のカーソルを指定することによってグラフを構築するような環境では，カーソルによって指定された位置から適切な式を決定する。次に，その式が実行された時刻の一覧を取得し，それを環境利用者に提示することによって時刻の指定を行わせる。

オブジェクトグラフの構築は，適切なオブジェクトをルートとして選択し，そこから到達可能なオブジェクトを順次訪問することによって行う。まずルートオブジェクトとして次を選択する。

1. グローバル変数のルートオブジェクト
2. 各ローカル変数に代入された値がオブジェクトの場合には，そのオブジェクト

ここから順次訪問を行う。

訪問した際には，そのオブジェクトが持つ各フィールドの履歴を全て確認し，指定された時刻における値を取得することによりオブジェクトグラフを構築する。フィールドの値がオブジェクトである場合には，オブジェクトグラフにそのオブジェクトに対応するノードを追加し，そのフィールド名をラベルとするエッジを新たに張る。その後，フィールドが指すオブジェクトが未訪問である場合には，再帰的に訪問を行う。フィールドの値がオブジェクトでない場合には，やはりオブジェクトグラフにその値を表すノードを追加し，そのフィールド名をラベルとするエッジを新たに張る。

2.2 Graal/Truffle

言語実現フレームワーク [18, 3] とは，効率的なメモリモデル，高性能なガーベージコレクタ，そして高速な JIT コンパイラを備えた仮想機械を生成する処理系である。PyPy [3] や TruffleRuby [17] などの処理系を実現するのに利用されており，いずれもオリジナルの処理系である CPython や CRuby と比較して高い実行性能を示している。

言語実現フレームワークを用いた処理系は，実現したい言語 (ゲスト言語) のインタプリタをフレームワーク言語 (ホスト言語) を用いて記述することにより実現される。フレームワークはこのインタプリタを受け取って仮想機械を生成する。

言語実現フレームワークの 1 つである Graal/Truffle は，GraalVM と呼ばれる仮想機械上でゲスト言語のインタプリタを実行時コンパイル (self-optimizing) する。これにより，ゲスト言語のプログラムを高速実行することができる。

2.2.1 SimpleLanguage

SimpleLanguage[2]は，Graal/Truffle 上で実現された動的型付き言語である。SimpleLanguage は Truffle フレームワークを用いた言語処理系のサンプルとして実現されている。そのため，小規模な言語ではあるものの以下に示される今日の実用的言語が共通して備える言語機能を有する。

1. 分岐と反復
2. 関数の定義と呼び出し
3. 関数内でのローカル変数
4. 動的オブジェクト

SimpleLanguage 処理系は，抽象構文木（abstract-syntax-tree，AST）インタプリタとして記述されている。言語定義を AST インタプリタの形式で実現することは Graal/Truffle からの要請である。AST におけるノードの定義は，Truffle フレームワークで定義された `Node` クラスのサブクラスを定義することによって実現する。このとき，各ノードが行うべき処理内容を `execute` メソッドに記述する。

具体的なノードの定義を整数値の加算を行う二項演算子を例にとって示す。整数値の加算を行う二項演算子ノード `IntAddNode` の実現例を図 2.7 に示す。このクラスは，その右辺ノードと左辺ノードを表すメンバ変数

[2] `https://github.com/graalvm/simplelanguage`

```
1  class IntAddNode extends Node {
2      Node lhs;
3      Node rhs;
4
5      int execute(Frame frame) {
6          int lhsResult = lhs.execute(frame);
7          int rhsResult = rhs.execute(frame);
8          return lhsResult + rhsResult;
9      }
10 }
```

図 2.7: 整数値の加算を行うノードの例

lhs 及び rhs を持つ。また execute メソッドでは，まず lhs 及び rhs の execute メソッドを呼び出すことで，右辺と左辺をそれぞれ評価する。次に，それぞれの評価値を足し合わせて，IntAddNode の評価値として返却する。ここで execute メソッドの引数 frame には，その式を評価する際の，ゲスト言語におけるスタックフレームが渡される。

第3章　提案手法

本章では，まず既存環境がどのように実現されているかを述べ，それぞれの実現方式において対象のプログラムに変更があった際に，どのような動作が行われるかを述べる。また，ライブプログラミング環境の特徴についても述べる。その後，本論文が提案する差分実行方式の概要を示す。

3.1　既存環境における問題点

3.1.1　既存環境の実行方式

デバッガ，及びライブプログラミング環境においては，プログラム実行して結果を表示するための方式は大きく分けて 2 つ存在する。

1. ブレークポイント方式
2. Back-in-Time 方式

ブレークポイント方式は，デバッグ開始以前にブレークポイントを設定しておき，デバッグが開始されるとブレークポイントまで実行を進め，その時点における実行時情報を提示する [4]。

Back-in-Time 方式を採用するデバッガは，プログラムが編集された際にプログラム全体を一度実行して実行時情報を記録しておき，その後環境利用者の要求に応じ実行時情報の履歴を遡り，その時点の情報を表示する。この方式は更に 2 つに分けることができ，実行を再開あるいは逆再生することが可能になるほどの，実行中の全ての情報を収集する Omniscient な方法 [8] と，デバッガが特に注目する情報に限って収集しておく方式 [6, 9, 13] がある。

3.1.2　プログラム更新時の挙動とその問題点

ライブプログラミングではプログラムが更新された際に速やかに表示を更新する必要があるが，既存環境では更新後にプログラムの大部分を再実行する必要が発生することが多いため，表示の更新に時間がかかる場

```
1 funcA();
2
3 funcC(); // Inserted line
4
5 funcB();
```

(A)

```
1 funcC(); // Inserted line
2
3 funcA();
4
5 funcB();
```

(B)

図 3.1: Omniscient Debugging における効率的な再実行が可能な例（A）と困難な例（B）

合がある。以下 2 つの方式について，どのように再実行が行われるかを述べる。

ブレークポイント方式では，プログラムが更新される度にプログラムを再実行する必要がある。これは，この方式では基本的にブレークポイント時点での実行時情報しか持たないためである。すなわち，プログラムの編集が発生したときに，その編集がブレークポイント時の状態に影響を与えるかを判定することができないため，編集後には必ず再実行が必要となる。

巻き戻し方式のうち，特に Omniscient デバッガの場合には，更新されたプログラム全体を再実行する代わりに，編集によってはじめて動作が変わる地点から再開させることができる。つまり，前回編集後の実行履歴を保持しておき，次に編集が発生しプログラムを実行し直す際に，編集によって動作を変えなければならない箇所（例えば編集によって新しく挿入された文）の直前の状態を，前回の実行履歴から再構築して，そこから実行を再開することができる。例えば図 3.1 に示した（A）のプログラムでは，編集によって 4 行目に新しい文が挿入されている。この場合，1 行目まで実行した後のスタック及びヒープの状態を再現することで，1 行目の `funcA` の実行を省略して再実行を行うことができる。

しかし，プログラムの変更がプログラムの先頭で発生しているような場合には，やはりプログラム全体を再度実行し直さなければならない。図 3.1（B）に示したプログラムの例では，編集によって 1 行目に文が挿入されている。この場合，編集によって影響がある地点はプログラム実行の最初となるから，プログラム全体を再実行し直す必要が発生する。

また巻き戻し方式のうちデバッガが特に注目する情報のみを収集する場合においては，プログラムの状態を全て記録するわけではないため，やはりブレークポイント方式と同様にプログラム全体の再実行が必要となる。

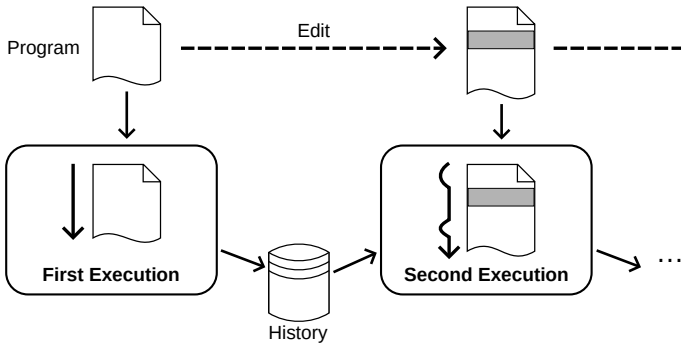

図 3.2: 差分実行の概略

```
1 let x = 1;
2
3 if (foo()) x = 2; // Inserted line
4
5 bar(x);
```

図 3.3: 静的解析では再実行省略の決定が困難なプログラム例

3.1.3　ライブプログラミング環境の特徴

ライブプログラミング環境において編集されたプログラムを再実行し直す際，実行しようとする編集後のプログラムと前回実行した編集前のプログラムの構文上の差異は小さいことが多い。これは，ライブプログラミング環境では，プログラムが編集されると即座にそのプログラムを再実行しようとするためである。したがって，コピー・アンド・ペーストや機械による複数ヶ所の同時的リファクタリング（例えば変数名の変更）などを除く，人力による編集では構文上の差異が小さくなりやすい。

3.2　差分実行方式の提案

本論文では，プログラムの編集があった際にその差分を前回実行時の実行時情報と併せて利用し，プログラム実行の一部分を省略しつつもオブジェクトグラフの可視化に必要な実行時情報を収集する，**差分実行方式**を提案する（3.2)。すなわち，プログラム実行時において編集の影響がある箇所とそうでない箇所を，前回実行時の履歴を用いながら動的に決定し，影響がない場所については前回の実行時情報を使いまわし，影響がある箇

所については実行し直すことで，編集後のプログラム実行の履歴を生成する。

この方法では，図 3.1 の（B）のようなプログラムでも，funcC の再実行のみで済ませることが可能となる。すなわち，funcC の呼び出しがそれ以降のプログラム実行に影響を及ぼさない場合には，funcA や funcB の呼び出しを省略できる。また funcC の呼び出しが以降に影響を与えるような場合においても，再実行すべき箇所を最小化することができる。

またこの方法では，再実行を行うかどうかを動的に決定するため，静的な方法よりも再実行が必要な箇所を減らすことができる。例えば図 3.3 で示したようなプログラムにおいて，5 行目の再実行の必要性を判断するにあたり，3 行目における変数更新の有無についての情報が必要となる。静的な方法では，foo の返り値が静的に決定できない限り 5 行目を再実行しなければならなくなるが，本手法では foo の呼び出しを実際に行いその返り値を得てから 5 行目の再実行を判定することができる。

ライブプログラミング環境においては，差分実行方式を採用することにより，編集後のプログラムの実行時情報を効率よく収集することができる。3.1.3 節で示したとおり，ライブプログラミングにおいては扱うプログラム列におけるそれぞれの差分は小さいことが期待される。編集差分が小さいほど実行時の影響も小さいと仮定すると，編集差分が小さいほど差分実行によって再度計算し直す必要のある箇所が減り，よって高速に実行時情報の収集ができるようになる。

第4章　差分実行方式によるプログラム実行

本章でははじめに，差分実行を用いた実行時情報収集手法の概略を述べる。次に，SimpleLanguage において，オブジェクトグラフの可視化のために必要となる，プログラム実行中に収集する必要のある履歴情報と，差分実行方式を実現するにあたり必要となる履歴情報を定義する。その後，新規実行中と再実行中それぞれの場合において，上 2 つの履歴を収集する方法を示す。また再実行中に前回実行されていない式や新たに挿入された式をどのように評価するかも併せて示す。

4.1　差分実行方式の概略

差分実行方式は，初めてプログラムが入力された際の新規実行と，プログラムの編集によって発生する 2 回目以降の再実行とで大きく実行の方式が異なる。これは，差分実行では編集前のプログラムの実行時情報を必要とするため，新規実行では前回実行時の履歴にあたる情報が存在しないことによる。

新規実行では，オブジェクトグラフ描画のための履歴情報と，差分実行を実現するために必要な履歴情報の両方を収集しながら，プログラム全体を通常の評価規則に従い実行する。実行中に収集される情報とは，各式の実行における，実行に用いた変数等の情報，実行結果，実行中に発生したオブジェクト更新などの副作用である。

一方再実行においては，前回実行時の履歴情報を利用することでプログラムの一部の実行を省略する。すなわち，編集によって実行に影響がある部分だけ再実行を行い，そうでない部分に関しては前回の履歴を使いまわす。

詳細には，ある式を再実行するにあたり，その実行中に用いられる変数等の環境が前回と同一であるならば，実行結果及び実行中に発生する副作用は前回実行と全く同一であるという仮定を置く。ここで，前回実行時の履歴には各式を実行する際に用いた変数等の履歴が記録されている。したがって，再実行中に前回と異なる状態となった変数を記録しておき，ある式を評価する際には，前回実行時におけるその式の評価中に用いた変数が

```
1  function bar(arg) {
2      let x = arg;              // nodeid = 1
3      return x;                 // nodeid = 2
4  }
5
6  function foo() {
7      let o = new object();     // nodeid = 3
8      o.baz = -1;               // nodeid = 4
9      return bar(o.baz);        // nodeid = 5
10 }
11
12 function main() {
13     let i = 1;                // nodeid = 6
14     foo();                    // nodeid = 7
15     while (x > 0) {           // nodeid = 8
16         bar(x);               // nodeid = 9
17         i = i - 1;            // nodeid = 10
18     }
19 }
```

図 4.1: nodeId が振られたプログラムの例

前回実行時と異なる状態を取っているか，すなわち編集による影響が及んでいるかどうかを確認することにより，その式の評価結果等が前回実行時と同一になるかを決定することができる。もし前回実行時と同じ実行結果等が生ずると判明した場合には，その式の実行を省略する。この省略の際には，その式の実行中に記録すべき履歴を，前回実行時に得られた履歴からそのまま複製して今回の履歴に記録する。これにより，その箇所の実行を省略しつつも履歴を補完することができる。また前回実行時と異なる実行結果が生じうる場合には，その式が持つ部分式に対して実行の省略が可能であるかの判定を行う。その後，部分式の実行結果を用いて元々実行しようとしていた式の評価を改めて行う。再実行中に編集によって挿入された式を実行しなければならない場合や，分岐において前回と異なる節の実行が必要となった場合には，その部分について新規実行を行う。

4.2　用語・記法の定義

本節では，以降用いられる用語と記法を定義する。

部分式

式 e の部分式を Child(e) で表す。部分式は直接に限らず子孫の式すべてを含む。

編集された式

編集後のプログラムにおいて，新たに挿入された式全体からなる集合を NewExps と書く。また，編集によって変更が発生した関数全体からなる集合を ModFuncs と書く。

式識別子

各式を識別するために，それぞれの式にユニークな識別子 nodeId を振る。この式識別子は編集によって変化しないように振り当てる必要がある。

nodeId が振られたプログラムの例を図 4.1 に示す。ただし，ここでは簡単のため式ではなく文ごとに nodeId を振ることとし，nodeId は具体的に整数値で表されているとする。また，このプログラムに編集があった場合にも，それぞれの文に振られた識別子は変更されない。

実行文脈

1 つの式が反復や関数呼び出しなどによって複数回呼び出されることがあるため，それらを区別するために**実行文脈**を定義する。これは従来の Kanon 実現におけるコールスタック（2.1.2 節参照）に対応するものである。具体的には実行文脈 execCtx は次のように定義される。

$$
\begin{aligned}
\langle \text{execCtx} \rangle &::= (\langle \text{callCtx} \rangle, \langle \text{nodeId} \rangle) \\
\langle \text{callCtx} \rangle &::= \langle \text{funcCall} \rangle :: \langle \text{callCtx} \rangle \mid \langle \text{loop} \rangle :: \langle \text{callCtx} \rangle \mid \epsilon \\
\langle \text{funcCall} \rangle &::= \texttt{Fc}\{\langle \text{nodeId} \rangle \textit{of the caller expression}\} \\
\langle \text{loop} \rangle &::= \texttt{Lp}\{(\langle \text{nodeId} \rangle \textit{of the loop expression}, \textit{loop count})\}
\end{aligned}
$$

ここで callCtx を**呼出文脈**と呼ぶ。呼出文脈は Kanon 実現における関数呼出文脈に対応し，コールスタックのように定義される。具体的には，関数を呼び出したときや反復が発生したときにプッシュされ，関数から抜け出したときや，反復が終了するとポップされる。関数の呼出は $\texttt{Fc}\{x\}$ のように表記され，x は適当な関数呼出式の式識別子が入る。反復は $\texttt{Lp}\{x, i\}$ のように表記され，x は反復全体を表す式であり，i にはその反復のカウントが入る。反復においては，反復カウントも併せて保持するため，イテ

レーションが終わるたびにポップとプッシュが発生する。また呼出文脈の基底 ϵ は，エントリポイント（ここでは `main` 関数）の呼び出しを表す。呼出文脈の中でも特に先頭の要素が関数呼び出しまたは基底であるようなものを 基礎呼出文脈 と呼ぶ。またある実行文脈 c が与えられたときに，その呼出文脈を 0 回以上ポップして初めて基礎呼出文脈 b が得られたとき，$\mathrm{Base}(c) = b$ と定める。例えば次が成り立つ。

$$\begin{aligned}
\mathrm{Base}((\epsilon, 6)) &= \epsilon \\
\mathrm{Base}((\mathtt{Fc}\{7\} :: \epsilon, 4)) &= \mathtt{Fc}\{7\} :: \epsilon \\
\mathrm{Base}((\mathtt{Lp}\{8, 0\} :: \epsilon, 10)) &= \epsilon
\end{aligned}$$

図 4.1 に示したプログラムの `bar` 関数に定義された返却文（$\mathrm{nodeId} = 2$）を例にとる。このプログラムにおいて当該文は 2 回実行される。初めて当該文が実行されるときの実行文脈は $(\mathtt{Fc}\{3\} :: \mathtt{Fc}\{5\} :: \epsilon, 2)$ となり，また 2 回目の実行における実行文脈は $(\mathtt{Fc}\{8\} :: \mathtt{Lp}\{6, 0\} :: \epsilon, 2)$ となる。

またこの実行文脈に対して実行順による全順序を入れる。この順序は式の実行終了の順番によって定まる。例えば図 4.1 で示したプログラムにおいて，$(\epsilon, 6) < (\mathtt{Fc}\{5\} :: \mathtt{Fc}\{7\} :: \epsilon, 1)$ や，$(\mathtt{Fc}\{5\} :: \mathtt{Fc}\{7\} :: \epsilon, 1) < (\epsilon, 7)$ が成り立つ。

実行文脈 c における実行について，その実行の途中に実行された式の集合を $\mathrm{VExps}(c)$ と書き，またその実行中に行われた実行の実行文脈の集合を $\mathrm{VCtxs}(c)$ と書く。$\mathrm{VExps}(c)$ には部分式や関数呼び出し先に存在する式の一部が含まれる。例えば以下が成り立つ。

$$\begin{aligned}
\mathrm{VCtxs}((\epsilon, 7)) = \{&(\mathtt{Fc}\{5\} :: \mathtt{Fc}\{7\} :: \epsilon, 1), (\mathtt{Fc}\{5\} :: \mathtt{Fc}\{7\} :: \epsilon, 2), \\
&(\mathtt{Fc}\{7\} :: \epsilon, 3), (\mathtt{Fc}\{7\} :: \epsilon, 4), (\mathtt{Fc}\{7\} :: \epsilon, 5), (\epsilon, 7)\}
\end{aligned}$$

差分実行方式において前回実行の存在が仮定されるとき，前回実行時に生じた実行文脈全体からなる集合を $\mathrm{PrevCtxs}$ と書く。

4.3　収集する必要のある実行時情報

本節では，差分実行方式を用いる，オブジェクトグラフ可視化が可能なライブプログラミング環境を実現するために収集する必要のある実行時情報を示す。収集する必要のある実行情報を大きく分けると，オブジェクトグラフ可視化のための実行時情報と，差分実行を行うための実行時情報の 2 つとなる。

実行文脈	内容
$(\epsilon, 6)$	$Var\{\epsilon, \texttt{i}\} \leftarrow \texttt{1}$
$(\texttt{Fc}\{7\} :: \epsilon, 3)$	$Obj\{(\texttt{Fc}\{7\} :: \epsilon, 3)\}$was generated
$(\texttt{Fc}\{7\} :: \epsilon, 3)$	$Var\{\texttt{Fc}\{7\} :: \epsilon, \texttt{o}\} \leftarrow Obj\{(\texttt{Fc}\{7\} :: \epsilon, 3)\}$
$(\texttt{Fc}\{7\} :: \epsilon, 4)$	$Fld\{(\texttt{Fc}\{7\} :: \epsilon, 3), \texttt{baz}\} \leftarrow \texttt{-1}$
$(\texttt{Fc}\{5\} :: \texttt{Fc}\{7\} :: \epsilon, 1)$	$Var(\texttt{Fc}\{5\} :: \texttt{Fc}\{9\} :: \texttt{Lp}\{8, 0\} :: \epsilon, \texttt{x}) \leftarrow \texttt{-1}$
$(\texttt{Fc}\{9\} :: \texttt{Lp}\{8, 0\} :: \epsilon, 1)$	$Var(\texttt{Fc}\{9\} :: \texttt{Lp}\{8, 0\} :: \epsilon, \texttt{x}) \leftarrow \texttt{1}$
$(\texttt{Lp}\{8, 0\} :: \epsilon, 1)$	$Var(\epsilon, \texttt{i}) \leftarrow \texttt{0}$

表 4.1: 図 4.1 のプログラムの実行によって得られる履歴

4.3.1　オブジェクトグラフ可視化のための履歴情報

2.1.3 章で示したとおり，オブジェクト可視化には次の履歴情報を収集する必要がある。

- 変数への代入
- オブジェクトのフィールドの初期化・更新
- オブジェクトの生成

これらは全て何らかの実行文脈において発生する情報であり，これらの情報は現在実行中の実行文脈全てと関連付けて記録される。変数への代入については，その変数のスコープも記録する。今回は呼出文脈を用いることでスコープを表す。オブジェクトの生成については，それぞれのオブジェクトを識別するために，オブジェクトと実行文脈を対応付けておく。この記録は他 2 つと異なり，実行後に破棄することができる。オブジェクトのフィールドの初期化・更新については，どのオブジェクトのどのフィールドにどの値が代入されたかを記録する。ここでオブジェクトの識別子として，そのオブジェクトが生成された実行文脈が記録される。

例として，図 4.1 で示されたプログラムを実行した際に得られる履歴を表 4.1 に示す。ここで $Var\{s, \texttt{var}\} \leftarrow val$ は s のスコープにおいて定義された変数 `var` に値 val が代入されたことを表し，$Fld\{ctx, \texttt{fld}\} \leftarrow val$ は実行文脈 ctx で生成されたオブジェクトのフィールド `fld` に値 val が代入されたことを表し，$Obj\{ctx\} \in val$ は実行文脈 ctx で生成されたオブジェクトへのポインタを表す。また変数の代入において代入先の変数の識別に呼出文脈も併せて用いているが，これはオブジェクトグラフ構築の際のルートとなる変数の選択に必要だからである。なお，この表における実行文脈は簡単のため 1 つしか示していないことに注意する。例えば，2

実行文脈	読み込み
$(\texttt{Fc}\{7\} :: \epsilon, 4)$	$Var\{\texttt{o}\}$
$(\texttt{Fc}\{7\} :: \epsilon, 5)$	$Var\{\texttt{o}\}$
$(\texttt{Fc}\{7\} :: \epsilon, 5)$	$Fld\{(\texttt{Fc}\{7\} :: \epsilon, 3), \texttt{baz}\}$
$(\texttt{Fc}\{5\} :: \texttt{Fc}\{7\} :: \epsilon, 1)$	$Arg\{\texttt{arg}\}$
$(\texttt{Fc}\{5\} :: \texttt{Fc}\{7\} :: \epsilon, 2)$	$Var\{\texttt{x}\}$
$(\texttt{Lp}\{8, 0\} :: \epsilon, 8)$	$Var\{\texttt{i}\}$
$(\texttt{Lp}\{8, 0\} :: \epsilon, 9)$	$Var\{\texttt{i}\}$
$(\texttt{Fc}\{9\} :: \texttt{Lp}\{8, 0\} :: \epsilon, 1)$	$Arg\{\texttt{arg}\}$
$(\texttt{Fc}\{9\} :: \texttt{Lp}\{8, 0\} :: \epsilon, 2)$	$Var\{\texttt{x}\}$
$(\texttt{Lp}\{8, 0\} :: \epsilon, 10)$	$Var\{\texttt{i}\}$
$(\texttt{Lp}\{8, 1\} :: \epsilon, 8)$	$Var\{\texttt{i}\}$

表 4.2: 図 4.1 のプログラムの実行によって得られる履歴

つ目の記録 $Obj\{(\texttt{Fc}\{7\} :: \epsilon, 3)\}$was generated は，$(\texttt{Fc}\{7\} :: \epsilon, 3)$ だけでなく，$(\epsilon, 7)$ とも関連付けて記録されている。

4.3.2　差分実行方式が要求する情報

前節で示した履歴情報に加えて，差分実行を行うために以下の履歴も収集する。

- 式の評価結果
- 式を評価する際に用いた引数，変数及びオブジェクトのフィールド
- 式を評価している途中に呼び出された関数

これらの情報も，前節と同様実行文脈と関連付けて記録する。式の評価に用いた変数等の記録について，オブジェクトのフィールドへの代入の際に左辺に変数やオブジェクトのフィールドが出現する場合があるが，これらも評価する際に用いたものであるとして記録する。

例として，図 4.1 で示されたプログラムを実行した際に得られる読み込みに関する履歴を表 4.2 に示す。ここで $Var\{\texttt{var}\}$ とは変数 `var` を読んでいることを表し，$Arg\{\texttt{arg}\}$ とは変数 `arg` を読んでいることを表し，$Fld\{ctx, \texttt{fld}\}$ は実行文脈 ctx で生成されたオブジェクトのフィールド `fld` を読んでいることを表す。前節では変数の識別に呼出文脈も併せて用いていたが，今回扱う SimpleLanguage においては，グローバル変数がないこと，及びある式の実行中に出現する変数は全て同一のスコープであることから，読み出しの記録には含めない。

実行文脈 c における実行において用いられた引数，変数及びオブジェクトのフィールド全体からなる集合を $\mathrm{Read}(c)$ と書く。

実行文脈 c における実行中に呼び出された関数全体からなる集合を $\mathrm{FuncCall}(c)$ と書く。例えば以下が成り立つ。

$$\mathrm{FuncCall}((\epsilon, 7)) = \{\texttt{foo}, \texttt{bar}\}$$

4.4　新規実行中の情報収集方法

新規実行においては，前 2 節で示した履歴を収集しながら通常通り実行を行う。

4.5　再実行中の情報収集方法

4.1 節で示したとおり，再実行時には前回実行時の履歴を用いて差分実行を行いつつ，再実行中の履歴情報を構築する。本節では，差分実行を行う際に必要となる，各変数・オブジェクトのフィールドが前回の実行と異なる可能性があるかを表す情報をまず定義する。次に再実行において，実行の省略を行う場合とそうでない場合にどのような処理を行うかを示す。その後，再実行中どのような場合にどのような処理によって式を実行するかを決定するアルゴリズムを述べ，最後に，具体的なプログラムを通して実際に差分実行における再実行がどのように行われるかを示す。

4.5.1　汚染フラグ

編集によって前回と異なる実行結果が発生したとき，その変化は以降のプログラム実行で伝播していくこととなる。各変数・オブジェクトに対して，編集による影響が発生しているか各変数・オブジェクトのフィールドが前回と異なる状態を取りうる場合には，その変数・フィールドに対して**汚染フラグ（dirty flag）**を立てることにする。汚染フラグが立つのは次の場合である。

1. 再実行中にある式を新規実行する場合において，その新規実行中に代入が行われた変数・フィールド
2. 再計算中に発生した変数・フィールドへの代入であって，代入される値が再計算または部分的な新規実行によって得られたものである場合の，その代入先の変数・フィールド

```
1  function corge(arg1, arg2) {
2      return 0;
3  }
4
5  function main() {
6      let x = 1;
7      let o = new Object();
8
9      x = 2;                          // Inserted line
10
11     o.qux = x;
12     let y = corge(o.qux, o);
13 }
```

図 4.2: あるプログラムに対して編集を行った例

また引数に対しても汚染フラグを定義する。引数の汚染フラグは，実引数を得るための実行がどの実行戦略（4.5.2 節参照）で行われたかによって定まる。もしも実引数が部分的な新規実行若しくは再計算によって行われたとき，その仮引数にフラグを立てる。引数の汚染フラグは関数呼び出しによって一部の引数に付与されたあと，新たにフラグが設定されることはない。

編集によってプログラムの一部が削除された場合には，その箇所で代入が発生した変数・フィールドに対して汚染フラグを立てる。

ある実行文脈 c の実行開始時点における，実行汚染フラグが立った変数，引数，フィールド全体からなる集合を Dirt と書く。

4.5.2　実行戦略

再実行中に実際に式をどのように実行し，その実行中の履歴をどのように取得するかという**実行戦略**は 3 つ存在する。

1. **再計算**: 部分式を再実行し，その実行結果をもとに再計算を行う
2. **履歴再生**: 実行の省略をし，履歴をそのままコピーする
3. **部分的な新規実行**: 新規実行し履歴を収集する

再計算

再計算の戦略では，部分式を再実行し，その結果をもとに式の結果を再度算出する。再計算中の履歴情報はそのまま記録される。この戦略は，部

分式やそれらが呼び出す関数の中に編集による変更箇所が存在する場合や，編集による影響がその式に及んでいる場合に採られる。以下，具体的な式を用いて説明する。

まず，部分式を持つ式の再計算の手順を述べる。例として加算を行う二項演算子 $e =$ `a + b` を考える。再計算においては，まず部分式である `a` と `b` について，再帰的に再実行を行う。この再実行中にはもちろん履歴も記録される。両部分式の再実行によって実行結果 v_1 と v_2 が得られたとすると，実際にこのあと $v_1 + v_2$ を計算し，これを e の実行結果として返す。またこの実行結果も部分式のとき同様履歴に記録される。また，オブジェクトのフィールドを読む式 $e =$ `r.f` においては，`r` と `f` を実行して実行結果としてそれぞれ o と k を得，その後オブジェクト r のフィールド f の現在の値を返却する。

次に，部分式を持たない式の再計算を考える。SimpleLanguage において部分式を持たない式は，リテラル式および変数・引数の読み込みを行う式の 2 つに限られるが，このうち再計算が発生しうる式は変数・引数の読み込みを行う式のみである。これは，4.5.3 節で述べられるアルゴリズムにおいて，部分式が存在しないことから $\mathrm{VExps}(c) = \varnothing$ が成り立つため $\mathrm{Dirt}(c) \cup \mathrm{Read}(c) \neq \varnothing$ が成立するが，リテラル式では常に $\mathrm{Read}(c) = \varnothing$ が満たされるためである。変数・引数の読み込みを行う式の再計算では，実際にその変数・引数の現在の値を返却され，それと同時に履歴に記録がなされる。

履歴再生

履歴再生では，その式を実際に実行することなく，前回の履歴を再生することにより代える。すなわち，前回実行においてその式を実行した際に記録された履歴をあたかも今回の実行でも発生したかのように記録し，また式の実行結果も前回と全く同じ値を返すようにする。この戦略は，ある式の実行が編集による影響を全く受けないような場合に採られる。

今回の実行についての履歴の反映は次のようにして行われる。ある実行文脈 c においてある式を履歴再生によって再実行を行う際には，前回実行時の履歴から実行文脈 $v \in \mathrm{VCtxs}(c)$ に結び付けられた記録を全て取り出し，そのまま今回の実行についての履歴に記録する。

部分的な新規実行

部分的な新規実行の戦略ではその式に限って新規実行を行う。新規実行中の履歴情報はそのまま記録される。この戦略は，編集によって挿入された式や，前回の実行で実行されなかった式に対して採られる。

Input　: expr: 判定すべき式を表すノード
　　ctx: 現在の実行文脈
Output: RE_CALC: 再計算
　　SKIP_EXEC: 履歴再生
　　NEW_EXEC: 部分的な新規実行

if expr $\in$ *NewExps* **or** ctx $\notin$ *PrevCtxs* **then**
　return NEW_EXEC
else if $\exists e \in$ *VExps*(ctx), *Child*(e) $\cup$ *NewExps* $\neq \varnothing$
　　or *Dirt*(ctx) $\cup$ *Read*(ctx) $\neq \varnothing$ **then**
　return RE_CALC
else
　return SKIP_EXEC
end

アルゴリズム 4.1: 実行戦略の決定

4.5.3　実行戦略の決定

差分実行方式での再実行においてある式を実行する際に，次の条件を確認しどの実行戦略によって実行を行うかを決定する。

1. その式は編集によって追加されたものであるか
2. その式が前回の実行において実行されているか
3. その式の実行中に編集によって新しく追加された式を実行する可能性があるか
4. その式の実行中に読み取る変数・引数・フィールドの中に汚染フラグが立っているものがあるか

詳細なアルゴリズムはアルゴリズム 4.1 に示したとおりである。

4.5.4　差分実行における再実行の例

本節では，図 4.2 に示されたプログラムを例にとり，このプログラムに対して実際に再実行を行う。これにより汚染フラグがいつ立つのかについてや，実行戦略がどのように決定されるかを示す。

まずはじめに，プログラム全体からなる式に対して再実行を試みる。このとき，プログラム全体からなる式は編集によって追加された式（9 行目）を部分式に持つから，再計算が行われる。

再計算によって，プログラム全体からなる式の直接部分式全てに対して再実行が行われる。ここではその直接部分式が main 関数の本体中の各文であるとする。

まず 6 行目及び 7 行目の文について，いずれもその式自身及びその部分式に編集によって追加された新しい式はなく，また汚染フラグが立っている変数・フィールドを用いていないから，両文については履歴再生をすることができる。

次に 9 行目の文について，この文は編集によって新しく追加されたものであるから，この文を部分的に新規実行する。9 行目の新規実行中には，変数 x への代入が発生する。このとき x に対して汚染フラグを付与する。

11 行目の文について，この文は編集によって構文上の影響を受けるわけではないが，前回実行では右辺の x を読んでいる。すなわち 11 行目の文を実行する際には汚染フラグが立った変数を読むことになるから，再計算を行い，o が指すオブジェクト（以下 Obj-A と記す）のフィールド qux に適当な値である 2 を実際に代入する。その後，o

12 行目の文について，まず関数 corge は編集による構文上の影響を受けていないとする。次に前回実行の履歴から，変数 o と，Obj-A のフィールド qux が読まれていることがわかる。ここで，Obj-A.qux には汚染フラグが付与されているから，この文は再計算をする必要があると判断する。部分式 o.qux は先程の理由により再計算をおこない，一方部分式 o には汚染フラグが立っていないことから，これは履歴再生を行い再実行とする。最後に関数を呼び出すが，関数呼び出し先の仮引数について，arg1 にのみ汚染フラグを付与し，corge の再実行を行う。corge の関数実行では，履歴再生によって直ちに値 0 を返す。その後代入が実行されるが，代入される値 0 は履歴再生によって得られたものであるから，変数 y には汚染フラグを付与しない。

以上によって図 4.2 に示したプログラムに対する差分実行による再実行が完了した。

第5章　実現

本章では，4 章で述べた実行方式を SimpleLanguage に実現した際の，詳細な実現方法について述べる。

5.1　時刻の導入

各式の実行について実行順に実行**時刻**を振り当て，履歴情報の一部を時系列として記録する。すなわち，前章で実行文脈と結びつけて記録されていた履歴情報の一部について，実行文脈ではなく時刻と結びつけて記録を行う。時刻を導入することにより，実行文脈と結びつけて記録する必要のあった情報をより簡潔に記録することを試みる。具体的には，変数・フィールドの読み書きに関する履歴情報について，これを時刻を用いて記録する。またこの他に，時刻と実行文脈との対応も記録する。

5.1.1　時刻の要件と実行文脈との対応付け

時刻とは次の操作ができ全順序をもつデータ構造である。

- （**インクリメント**）：時刻 t が与えられたときに，その時点において存在する任意の $s > t$ を満たす時刻 s に対して $t < t' < s$ を満たすような時刻 t' を返す
- （**中間時刻の生成**）：時刻 t_1 と $t_2 > t_1$ が与えられたときに，$t_1 < t < t_2$ を満たす時刻 t を返す

ここで時刻 t に対してインクリメントの操作をして得られた時刻を $\mathrm{inc}(t)$ と書き，また t_1 と $t_2 > t_1$ の中間の時刻を $\mathrm{mid}(t_1, t_2)$ と書く。

前節で述べたとおり，時刻は実行文脈との対応を持つ。すなわち，ある実行文脈のもと行われる実行について，その実行開始時刻と終了時刻が記録される。実行文脈について，式識別子は編集前後で変更がない式に対しては同じ識別子が与えられることから，同じ実行文脈においては同じ時刻を振り当てる必要がある。

時刻は各式の実行に対して次のように割り振られ，また履歴に記録される。

- 初回実行の新規実行の前に，現在時刻として t_0 を設定する。
- 現在時刻が t であるときに実行文脈 c において式 e を新規実行したとする。このとき，
  1. まず c の実行開始時刻を t として記録する。
  2. e の実行中に他の式を実行する必要がある場合には，それを実行する。
  3. 変数・オブジェクトの読み書きなどで履歴に時刻を記録するときには，その時々の現在時刻を記録する。
  4. e の実行が終了した直後に，c の実行終了時刻として，そのときの現在時刻 t' を記録する。その後，現在時刻を $\mathrm{inc}(t')$ とする。
- 再実行については次のようにする。
  1. 再計算および履歴再生では前回と同じ時刻を割り振る。
  2. 部分的な新規実行においては，次のようにする。ここでは実行文脈 c において式 e について部分的な新規実行を行うことを考える。
     (i) 部分的な新規実行の前に，現在時刻として $\mathrm{mid}(t_1, t_2)$ を設定する。ただし，t_1 とは $d < c$ を満たす最大の実行文脈 d の終了時刻であり，また t_2 とは前回実行において記録された時刻のうち，$t_2 > t_1$ を満たす最小の時刻である。もしも d が存在しない場合には，$t_1 = [-1]$ とする。また t_2 が存在しない場合には，現在時刻を $\mathrm{inc}(t_1)$ とする。
     (ii) 以下新規実行の場合と同様に実行を行う。

5.1.2　時刻の具体的な実現

本実現では時刻は可変長の整数配列を用いて表現する。具体的には，まず初回実行時の最初に設定される現在時刻について $t_0 = [0]$ とする。inc 関数について，この関数は受け取った配列の末尾要素をインクリメントした配列を生成して返却することにより実現される。例えば以下が成り立つ。

$$\begin{aligned} \mathrm{inc}([1]) &= [2] \\ \mathrm{inc}([1,2,3,4]) &= [1,2,3,5] \end{aligned}$$

Input　: $t_1 = [a_0, \ldots, a_{n-1}]$, $t_2 = [b_0, \ldots, b_{m-1}]$
Output: $t_1 < t < t_2$ を満たす時刻 t

if $n > m$ **then**
　$t \leftarrow$ new array of length $m + 1$
　for $i \leftarrow 0, \ldots, m - 1$ **do** $t[i] \leftarrow a_i$
　$t[m] \leftarrow a_m + 1$
else
　$l \leftarrow \texttt{mismatch}(t_1, t_2)$
　$t \leftarrow$ new array of length $l + 2$
　for $i \leftarrow 0, \ldots, l - 1$ **do** $t[i] \leftarrow a_i$
　if $l = n$ **then**
　　$t[l] \leftarrow b_l - 1$
　　$t[l + 1] \leftarrow 0$
　else if $l = n - 1$ **then**
　　$t[l] \leftarrow a_l$
　　$t[l + 1] \leftarrow 0$
　else
　　$t[l] \leftarrow a_l$
　　$t[l + 1] \leftarrow a_{l+1} + 1$
　end
end

アルゴリズム 5.1: 中間時刻の生成

次に，中間時刻の生成を行うアルゴリズムをアルゴリズム 5.1 に示す。このアルゴリズムを用いた例を以下に示す。

$$
\begin{aligned}
\mathrm{mid}([1], [4]) &= [3, 0] \\
\mathrm{mid}([1, 2], [1, 3]) &= [1, 2, 0] \\
\mathrm{mid}([1, 0], [1, 0, 0]) &= [1, -1, 0] \\
\mathrm{mid}([1, 3, 5], [2, 3, 5, 7]) &= [1, 4] \\
\mathrm{mid}([1, 0], [3]) &= [1, 1]
\end{aligned}
$$

Input : $a = [a_0, \ldots, a_{n-1}]$, $b = [b_1, \ldots, b_{m-1}]$

$l \leftarrow \min(n, m)$
for $i \leftarrow 0, \ldots, l-1$ **do**
  if $a_i = b_i$ **then return** i
end
if $n = m$ **then return** -1
else return l

アルゴリズム 5.2: Mismatch

また，順序は辞書順によって定める。例えば，次が成り立つ。

$$\begin{array}{lcl} [1] & < & [2] \\ [0,0,0,0] & < & [0,0,1,0] \\ [0,0,0] & < & [0,0,0,1] \end{array}$$

5.2　収集する必要のある実行時情報と履歴の構造

本節では 4.3 節で示した，差分実行を行う，オブジェクトグラフの可視化が可能なライブプログラミング環境が収集する必要のある実行時情報について，これらの情報を実現において実際にどのように格納しておくかについて述べる。

5.2.1　実現で用いるデータ構造

本節では，実現中で用いた型及びデータ構造を定める。
まず一般的なデータ構造は次のとおりである。

- `(A, B,...)`: タプルを表す。`A` や `B` は要素の型である。
- `List<T>`: 可変長の配列である。`T` は要素の型である。
- `Stack<T>`: スタックを表す。`T` は要素の型である。
- `Map<K, V>`: キーに値をマッピングできる構造である。 `K` はキーの型であり，`V` は値の型である。
- `Set<T>`: 集合を表す。`T` は要素の型である。

- TImeList<T>: 時刻をキーとて値をマッピングできる構造である。Map<時刻，V> との違いとして，内部で時刻に基づき常にソートされていることが挙げられる。V は要素の型である。

代入発生の記録などで SimpleLanguage 上の値を記録をする必要がある場合，その値をそのまま記録するのではなく別の方法で表現した値を記録する。これは，代入発生を記録する際に，代入された値として SimpleLanguage 上のオブジェクトをそのまま記録してしまうと，前回実行時に生成されたオブジェクトが履歴に残り続けてしまい，GC に回収されなくなるためである。本実現では代入などで SimpleLanguage 上の値をそのまま記録する代わりに，履歴に残しても良い値（例えば，数値や文字列）についてはその値をラップしたものを記録し，オブジェクトについてはそのオブジェクトが生成された際の実行文脈を記録する。このように表現された SimpleLanguage 上の値を表す型を SLV と書く。

ただし，一部の履歴では SimpleLanguage 上の値を一時的にそのまま保存する必要がある。ここでは，SimpleLanguage 上のオブジェクトを指す型を SLO と書く。

ある式の実行中に発生した変数・引数・フィールドの読み込みを記録する際には，その変数・引数・フィールドを識別する必要がある。ここでは，変数・引数・フィールドを特定する値の型を SLE と書く。

5.2.2　オブジェクトグラフ可視化のための履歴情報

まず時刻と実行文脈の相互変換が可能となるように，その対応を記録するデータ構造を定義する。具体的には次のデータ構造を用いて定義した。

- timeToCtx: TimeList<実行文脈>
  - 時刻から実行文脈への変換を記録する。この時刻とは，その実行文脈の実行終了時刻である。
- ctxToTime: Map<実行文脈，(時刻，時刻)>
  - 実行文脈から時刻への変換を記録する。ハッシュマップの値は，キーとなっている実行文脈の実行開始時間と終了時間からなるタプルである。

オブジェクトグラフ可視化のための履歴情報について，本実現では次のように定義した。

- varUpdates: Map<基本呼出文脈，Map<変数名，TimeList<SLV>>>

  - ローカル変数の値の履歴を，スコープ及び変数名ごとに記録する。

- objToGenTime: Map<SLO，時刻>
  - SimpleLanguage プログラム上のオブジェクトとその生成時刻を記録する。SimpleLanguage プログラムでオブジェクトが生成されるたびにエントリが追加される。この情報は 4.3.1 節で示したとおり，実行が完了したあとに破棄して良い。また，objIdToGenCtx にてキーとして保管されているある SimpleLanguage 上のオブジェクトについて，そのオブジェクトが現在実行中の SimpleLanguage プログラム上で辿れなくなった場合には，そのオブジェクトをキーとするエントリを削除して良い。

- fldUpdates: Map<実行文脈, Map<フィールド名, TimeList<SLV>>>
  - フィールドの値の履歴を，スコープ及び変数名ごとに保存する。外側の Map のキーは，オブジェクト生成時の実行文脈を表す。

5.2.3　差分実行方式が要求する履歴情報

差分実行方式が要求する履歴情報について，本実現では次のように定義した。

- execResults: TimeList<SLV>
  - 式の評価結果を記録する。

- readVars: Map<基本呼出文脈，TimeList<変数名>>

- readArgs: Map<呼出文脈，TimeList<引数名>>

- readFlds: Map<実行文脈，TimeList<フィールド名>>
  - 式実行中に発生した変数・引数・フィールドそれぞれの読み込み履歴を記録する。

- calledFunctions: TimeList<(基本呼出文脈，関数名)>
  - 関数呼び出しが発生した際に，その関数名と，関数呼び出し後の基本呼出文脈を記録する。

5.3　新規実行中の情報収集方法

新規実行においては，時刻や実行文脈を適切に更新しながら通常通り式を実行し，履歴を収集する。具体的には，以下に示す場合に履歴の記録を行う。

式の実行

まず式を実行する前に，その時点における時刻を実行開始の時刻としてメモしておく。式の実行が終了した際には再度時刻を取得し，実行文脈と共に `timeToCtx` の末尾に追記する。また，メモしておいた開始時刻を取り出し， `ctxToTime` に記録を追加する。さらに，式の実行結果を終了時刻とともに `execResults` の末尾に追記する。

オブジェクトの生成

オブジェクトの生成が発生した際には，発生の時刻と共に `objToGenTime` 及び `genTimeToObj` に記録する。

変数・引数・フィールドの読み込み

変数・引数・フィールドの読み込みが発生した際には，その対象を現在時刻とともに `readEntities` の末尾に追記する。

変数・引数・フィールドの書き込み

変数・引数・フィールドの読み込みが発生した際には，`varUpdates` や `fldUpdates` から適切な `TimeList` を取り出し，その末尾に追記を行う。ただし，書き込まれる値がオブジェクトである場合には，オブジェクトをそのまま生成するのではなく，`objToGenTime` を用いてそのオブジェクトが生成される時刻を取り出し，その値を代わりに記録する。

関数呼び出し

関数呼び出しが発生した場合には，その関数名と関数呼び出し後の基本呼出分脈を `calledFunctions` の末尾に記録する。

5.4　再実行中の情報収集方法

本節では，具体的にどのようにして再実行を行い，またそれと同時に実行時情報を収集するかを述べる。まず，効率よく再実行・実行時情報収集を行うための技法をいくつか紹介する。次に，4.5.3 節で示した実行戦略を決定するアルゴリズムについて，5.2.1 節で説明したデータを用いて実際にどのように実現されているかを説明する。最後に，各実行戦略においてどのようにプログラムを再実行するかを述べる。

5.4.1　効率化のための特殊な技法

まず，再実行においては，変数・フィールドの代入やオブジェクトの生成を遅延させる。また，部分的な新規実行の戦略により再実行を行う際には，その途中で記録される実行時情報を，一時的に別の場所に溜めておき，新規実行終了後に元の履歴への併合を行う。本節ではこれらの技法の詳細を紹介したあと，再実行の具体的な処理を示す。

代入・オブジェクト生成の遅延

単純な再実行の実現では，変数・フィールドへの代入や，オブジェクトの生成は即座に行われる。すなわち，再実行中に履歴再生の戦略が採られた場合には，前回の履歴から再生すべき範囲を取り出して，その内に記録された全ての代入やオブジェクト生成を逐次実行し，また再計算の戦略が採られた際に代入やオブジェクト生成を行う必要が発生した場合には，実際にその代入やオブジェクト生成を実施して，その後履歴に書き加えることをする。しかしこのような実現の場合，実行を省略できる箇所を実行し直すのと何ら変わらず，したがって差分実行の利点が消滅することになる。

再実行においては，変数・フィールドの代入やオブジェクトの生成を可能な限り遅延させる。すなわち，履歴再生の戦略においては，代入やオブジェクト生成の履歴を読み取らず，その式の前回の実行結果を履歴から取得して返すことだけを行い，また再計算の戦略において代入やオブジェクト生成の必要が生じた場合には，履歴にそれらを記録するだけとし，実際の操作は行わない。

これらの遅延は，部分的新規実行を行う際に解決する。すなわち，部分的新規実行を行う場合に，その直前でその時点におけるオブジェクトグラフ及びローカル変数環境の構築を履歴をもとに構築する。これは，新規実行においては通常の実行と同様に，代入やオブジェクトの生成が即座に行われることを想定しているためである。

部分的な新規実行の際のの履歴記録

部分的新規実行では，履歴の記録にコストがかかる場合がある。5.2.2 節や 5.2.3 節に記したとおり，履歴情報の多くは TimeList の構造を用いて記録されている。5.2.1 節で示したように， TimeList は中の要素をキーである時刻によって常に整列されている。部分的な新規実行は，プログラム実行の途中で発生するから，部分的新規実行において履歴に記録を挿入する際に，整列を保つためにコストがかかる。このコストは履歴に記録を挿入するたびに発生するため，効率に関して問題がある。

本実現では，部分的な新規実行の際に記録を一度別のところにためておいて，新規実行が終了したところで元の履歴に一気に挿入することにより，このコストを削減することを試みる。新規実行中は履歴情報を一切用いずに実行を行うため，本技法を採用しても実行に問題が生じることはない。具体的には，部分的な新規実行を行う直前で空の履歴を用意し，新規実行中にはこの履歴に記録を行う。新規実行が終了したあと，新規実行中に記録が行われた履歴を，元の履歴に併合する。ここで，TimeList の併合に関しては，元の履歴のある 1 点に新規実行中の履歴を全て挿入することになる。これは，部分的な新規実行が前回実行でも行われた 2 つの隣接する実行の間に発生するためである。

5.4.2　再実行中に用いる一時データ

再実行中には，5.2 節で述べたデータの他に差分実行を行うために必要なデータを保持する。このデータは 5.2 節で定義したデータとは異なり，実行後破棄して良い。

再実行のために必要な一時データは次のとおりである。

- dirtyVars: Stack<Set<変数名>>
- dirtyArgs: Stack<Set<引数名>>
- dirtyFlds: Map<実行文脈，Set<フィールド名>>>
  - それぞれ汚染フラグが立った変数・引数・フィールドの集合である。4.5.1 節で示した Dirty に対応する。dirtyVars と dirtyArgs のスタック構造は，関数スタックに対応する。即ち，関数が呼び出されるたびに dirtyVars と dirtyArgs に空の Set がプッシュされ，関数から抜け出すたびに dirtyVars と dirtyArgs がポップされる。
- genTimeToObj: Map<時刻，SLO>

- `objToGenCtx` の逆対応であり，オブジェクトの生成時刻からオブジェクトを逆引きするために記録する。用途については 5.4.4 節にて説明する。`objToGenCtx` と同様に，現在実行している SimpleLanguage プログラム上で辿れなくなったオブジェクトを値として持つエントリはいつでも削除して良い。

5.4.3　実行戦略決定アルゴリズムの実現

4.5.3 節で示した実行戦略を決定するアルゴリズムが，5.2.1 節で定義したデータを用いてどのように実現されているかを述べる。具体的には，アルゴリズム 4.1 に示されたアルゴリズムの，それぞれの条件式の実現を述べる。これから実行しようとする式を `expr` とし，また実行文脈を `ctx` と書く。

- $\texttt{ctx} \notin PrevCtxs$
  - `ctxToTime` に `ctx` に対応するエントリが存在するかどうかによって判断できる。
- $\exists e \in \mathrm{VExps}(\texttt{ctx}), \mathrm{Child}(e) \cup \mathrm{NewExps} \neq \varnothing$
  - まず `expr` が編集によって追加された部分式を持っているかどうかを判定し，もし持っていないときには，`ctxToTime` を用いて `ctx` に対応する時刻 (t_1, t_2) を求め，`calledFunctions` を用いて (t_1, t_2) の間に呼び出された関数を全て求め，その関数に新しい式が含まれているかで判定できる。
- $\mathrm{Dirt}(\texttt{ctx}) \cup \mathrm{Read}(\texttt{ctx}) \neq \varnothing$
  - `dirtyVars` と `readVars`，`dirtyArgs` と `readArgs`，および `dirtyFlds` と `readFlds` をそれぞれ突き合わせることで判定できる。具体的には，`ctxToTime` を用いて `ctx` に対応する時刻 (t_1, t_2) を求め，`readVars`，`readArgs`，`readFlds` から (t_1, t_2) の間の履歴を取り出し，その間に読み出された変数･引数･フィールドの中に `dirtyVars`，`dirtyArgs`，`dirtyFlds` に含まれるものが存在するかで判定できる。

5.4.4　再実行の実現

本節では 5.4.1 節で示した技法を用いて，どのように再実行が実現されているかについて，実行戦略および式ごとにその詳細を述べる。

```
1 let x = true;
2 x = false;       // Inserted
3 if (x) {
4     foo();
5 } else {
6     bar();
7 }
```

図 5.1: 編集によって分岐先が異なるプログラムとその編集の例

再計算

4.5.2 節に示したとおり，自身の部分式を再実行し，その実行結果を用いて適切に計算し直し，その結果を返すと同時に履歴を書き換える。すなわち，時刻 t において式 e を再計算によって再実行する場合には，e の部分式を再実行した上でその部分式の実行結果をもとに e の新しい実行結果 v を算出し直し，これを返す。またそれと同時に，`execResults` の t に対応する値を v に書き換える。e が代入式である場合には，`varUpdates` や `fldUpdates` の適切な箇所を書き換える。v が時刻 u に生成されたオブジェクトを表していた場合には，`genTimeToObj` から u に対応する適当なオブジェクトがすでに存在するかを確認する。存在する場合にはそのオブジェクトを返し，存在しない場合には SimpleLanguage 上の空オブジェクト o を生成し，`objToGenTime` と `genTimeToObj` に適切に記録を行った上で，o を返す。

また式を実行して値を返すときには，その値がどの戦略によって得られた値であるかも同時に返す。この情報は代入式を実行する際に，代入先の変数・フィールドの汚染フラグを立てるかどうかの判断に用いる。即ち 4.5.1 節に示したとおり，代入式によってある変数・フィールドに値が代入されるとき，その値が履歴再生によって得られたのであればフラグを立てず，再計算あるいは部分的な新規実行によって得られた場合には，その代入先の変数・フィールドに汚染フラグを立てる。

実現において特別な扱いをする式として，変数・引数・フィールドの値を読み出す式，及び分岐が発生する式が挙げられる。

変数・引数・フィールドの値を読み出す式においては，5.4.1 節に示したとおり `varUpdates` や `fldUpdate` から値を取り出す。これは，5.4.1 節で述べたとおりオブジェクト等を含む環境の構築が遅延されているためである。

`if` 文や `while` 文など分岐が発生する式では，履歴の一部分を削除する必要が生じる場合がある。これは，本実現では編集前の実行の履歴を書き換えることにより編集後の実行の履歴を得るから，編集前後で異なる分岐

節が選択された際に履歴中の適当な箇所を削除しないと，両方の実行が履歴に保存され続けるためである。例えば，図 5.1 で示すプログラムとその編集では，初回の実行では `foo` が実行されるものの，編集後の実行では `bar` が実行される。ここで適切に履歴を削除しないと，編集後の実行履歴中に編集前の実行で発生した `foo` における実行時情報が残り続けることとなる。このような事態を避けるため，`if` 文で分岐が発生した場合にはその分岐で実行されない方の履歴を削除する必要がある。また `while` 文についても，反復が n 回で終了した場合，履歴に $n+1$ 回以降で記録された情報が残留していないかを確認し，存在する場合には削除する必要がある。また，SimpleLanguage は早期リターンや `continue` 文を持つから，これらに対しても適切な処理を実現しておく必要がある。

履歴再生

実行文脈 c で表される実行を履歴再生で行うと判断された場合には，`execResults` から c の実行結果 v を取り出しその値を返却する。v がオブジェクトを表している場合には，再計算の際と同様に `genTimeToObj` 等を用いて適当なオブジェクトを取得・生成し，それを返す。

部分的な新規実行

部分的な新規実行を行う際には，まず初めに 5.4.1 節にあるとおり変数環境及びオブジェクト生成の遅延を解決する。呼び出し文脈 c において，ある式の部分的新規実行を行う際には，具体的に次のようにして遅延を解決し環境を構築する。なお，部分的な新規実行を行う直前に行われた式実行の時刻を t と書くことにする。

- **（生成済みオブジェクトの同期）** `objToGenTime` にオブジェクトが存在する際には，そのオブジェクトが持つ各フィールドの値を更新する。`objToGenTime` に記録されているオブジェクトは，再計算や履歴再生の際にプレースホルダ的に生成されていたり，あるいは，前回の部分的な再実行によって生成されたりしたものである。本手法では部分的な新規実行を行わない限り，オブジェクトのフィールドの追加・値の更新・削除を行わないから，新規実行を行う前に予めオブジェクトの状態を更新しておく必要がある。具体的には，`objToGenTime` に記録されたオブジェクト p とその生成時間 r を取り出し，下記の生成を遅延したオブジェクトの復旧の手順と同様にして行う。ただし同手順中においては，新しいオブジェクトを生成せず，o を p に，u を r に読み替える。

- **（引数・変数の同期）**c の時点において辿ることのできるローカル変数・引数をすべて取り出す。まず `varUpdates` からキー $\mathrm{Base}(e)$ に対応する `Map` を取り出し，それが持つ各変数の履歴に対して 時刻 t における値を取り出し，環境に反映する。もしもその値がオブジェクトを表していた場合には，次の生成を遅延したオブジェクトの復旧手順に従いオブジェクトを生成・復旧し，そのオブジェクトを環境に反映する。

- **（生成を遅延したオブジェクトの復旧）**未達であった，時刻 u に生成されたオブジェクトが発見されたとき，新しくオブジェクト o を生成する。次に，`objToGenTime` と `genTimeToObj` に u と o のペアをそれぞれ適切に記録する。その後，`fldUpdates` から o の各フィールド f の値の履歴を取り出し，時刻 t における値 v を求め，o のフィールド f に値 v を設定する。ここで，v が時刻 s に生成されたオブジェクトを表していた場合には，`genTimeToObj` から s に対応する適切なオブジェクト q を取り出して v に代えてフィールドの値として設定する。もしも s に対応するオブジェクトが存在しなかった倍には，本手順を s について再帰的に適用し，そこで生成されたオブジェクトを v に代えてフィールドの値として設定する。

環境の構築が完了したあとは，5.3 節に従いその式を新規実行する。

また，4.5.1 節に示したとおり，部分的な新規実行中に代入が発生した変数・フィールドについては，それぞれにフラグを立てる。

第6章　評価

本論文で提案する差分実行方式によるプログラム実行が，実際のプログラムとその編集に対してどれほど効率的に履歴を収集できるかについて評価を行う。具体的には，あるプログラムに対して何らかの編集を加え，編集前後における履歴収集のための実行にどれほど時間がかかるかで評価をする。

6.1　評価対象

本論文では，差分実行方式が特に有効になりうるプログラム及び編集例に対して評価を行い，実際に有効であるかを確かめた。差分実行方式が有効となると考えられるプログラム及び編集例は次のとおりである。

- プログラムの編集によって前回と異なる実行が発生する箇所が極めて小さい場合
- プログラムで扱うデータ構造が複雑であるなど，データの構築に時間がかかる場合

評価の対象として図 6.1 に示すようなプログラム及び編集例を AVL 木及び連結リストそれぞれに実現し，編集前後のプログラム実行時間をそれぞれ計測した。このプログラムは，キーと値をペアとして保持するデータ構造に対して要素を追加し，その後データ構造内のすべてのキーの総和をとる。プログラム中の `new Holder()` とはそれぞれのデータ構造のルートオブジェクトの生成を行い，関数 `add` はデータ構造への追加を行い，関数 `sumOfKey` は与えられたデータ構造がもつ要素のキーの総和をとる。関数 `add` について，AVL 木ではキーの順序を保つ一方で，連結リストでは単に先頭にキーと値のペアを格納する。AVL 木と連結リストを比較することで，データ構造の複雑さが実行時間にどれほど影響を与えるか評価する。また，`CHANGED` と `CALC_LOOP` の値はベンチマーク前に定めるパラメタである。`CHANGED` にいくつかの値を設定し，編集後プログラムの再実行における再計算が必要となる式の数を変更することで，再計算のコストがどれほどであるかを比較する。また，`CALC_LOOP` にいくつか

```
function main() {
    let tmp = new Holder();
    let label = "Label1";
    let i = 0;
    for (; i < 500 - CHANGED; i++) {
        add(tmp, i, label)
    }
    label = "Label2";   // Inserted line
    for (; i < 500; i++) {
        add(tmp, i, label)
    }

    for (i = 0; i < CALC_LOOP; i++) {
        sumOfKey(tmp);
    }
}
```

図 6.1: ベンチマーク対象プログラム

CPU	AMD Ryzen 9 5900X
メモリ	64GB RAM (DDR4-3200)
OS	Linux 6.1.1-gentoo
JVM	OpenJDK 1.8.0_302 (build 25.302-b06-jvmci-21.3-b02)

表 6.1: 評価環境

の値を設定することで，編集後プログラムの再実行における戦略決定と履歴再生のコストがどれほどであるかを比較する。

本論文では，上記の編集前及び編集後のプログラムについて，次の処理系で実行しその実行時間を比較する。

- 何も改変を加えていない SimpleLanguage 処理系（グラフ中の **Original**，以下同様）
- Kanon が必要とする履歴情報を通常の実行方式によって収集する SimpleLanguage 処理系（**Record Only**）
- 本論文で提案した，Kanon が必要とする履歴情報を差分実行方式を用いて収集する SimpleLanguage 処理系（**Diff Exec**）

本評価は表 6.1 に示した環境にて行った。

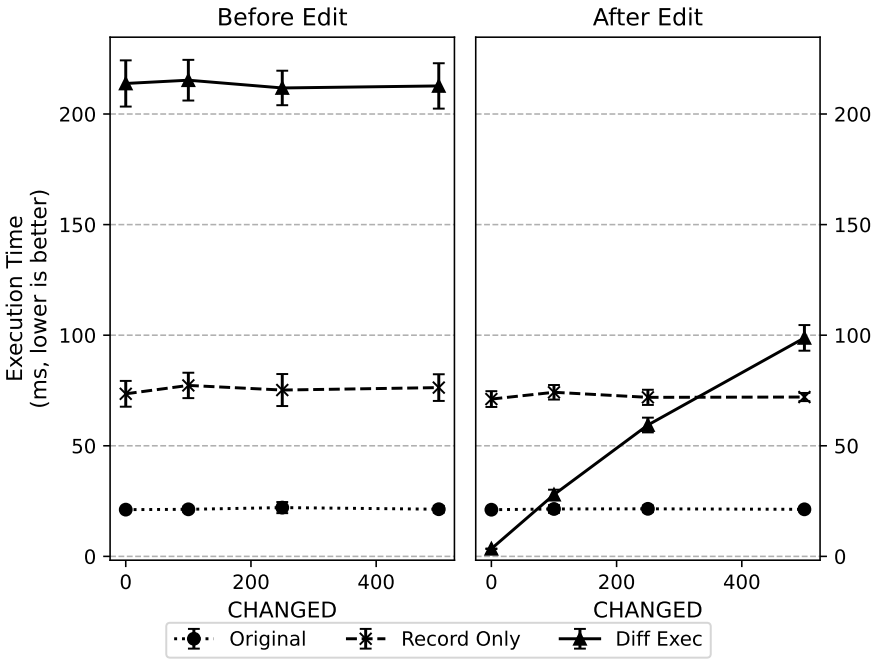

図 6.2: AVL 木を用いた場合の実行時間（CALC_LOOP = 5）

6.2　評価結果

ベンチマークの結果を図 6.2，6.3，6.4，6.5 にそれぞれ示す。

まず，CALC_LOOP 一定のまま CHANGED の値を変更した場合，すなわち編集の影響を受ける箇所を変化させた場合を確認する（図 6.2，6.4）。差分実行をしない 2 つの処理系においては，挿入される要素の量は編集による影響を受けないから，編集の有無や CHANGED の値によらず実行時間は一定であったと言える。またオブジェクトグラフ収集を行う処理系の実行時間は，改変前の処理系と比較し 3-6 倍程度だった。差分実行を行った場合における編集前の実行時間は，オブジェクト可視化用の履歴のみを収集する処理系の 3 倍程度であった。編集後の実行時間は CHANGED の値に比例しており，編集の影響が小さい場合には，改変を加えていない処理系に匹敵するほどの実行時間であるが，編集により再計算される式の数が増加すると，オブジェクトグラフ収集を行う処理系よりも悪化した。

続いて，CHANGED 一定のまま CALC_LOOP の値を変更した場合，すなわち履歴再生により再実行される式の数が増加した場合を考える（図 6.3，6.5）。差分実行をしない 2 つの処理系においては，CALC_LOOP の値に比例した。またオブジェクトグラフ収集を行う処理系の実行時間は，改変前の処理系と比較し 2-6 倍程度であった。差分実行を行った場合における編集

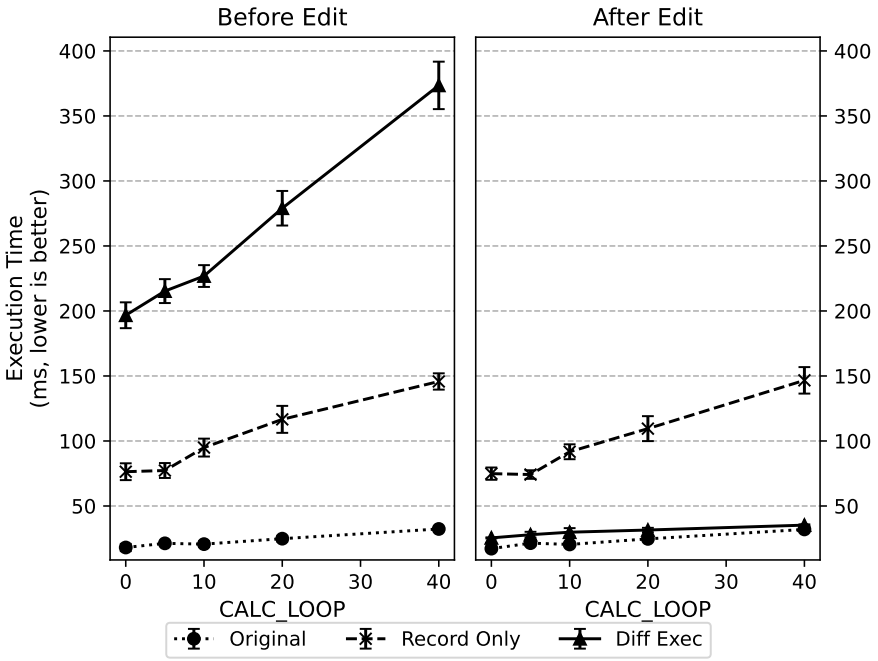

図 6.3: AVL 木を用いた場合の実行時間（CHANGED = 100）

前の実行時間は，CHANGED を変更した場合と同様，オブジェクト可視化用の履歴のみを収集する処理系の 3 倍程度であった。編集後の再実行時間は CHANGED の値に比例するが，全ての場合でオブジェクト可視化用の履歴のみを収集する処理系と同等かそれよりも高速であった。

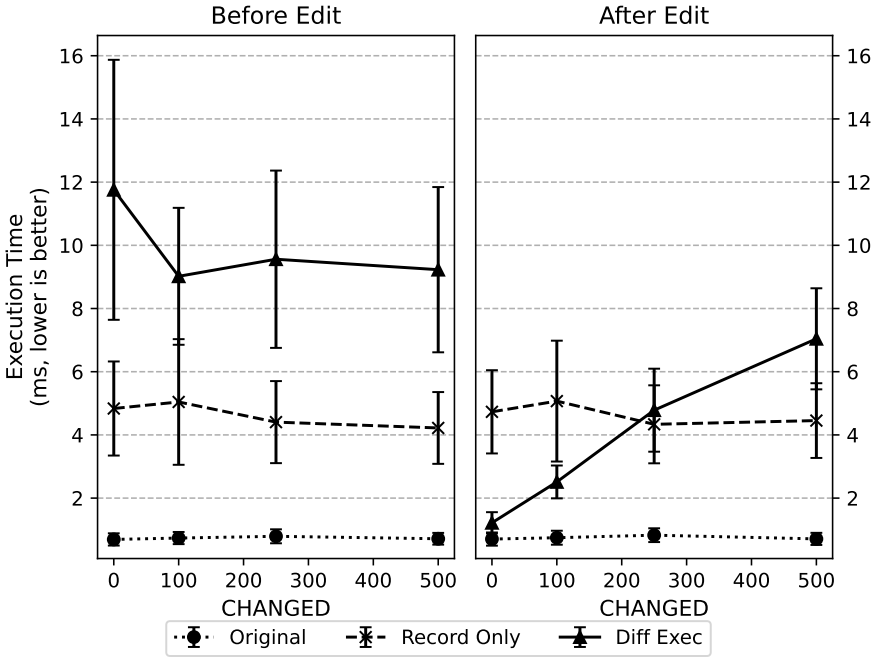

図 6.4: 連結リストを用いた場合の実行時間（CALC_LOOP = 5）

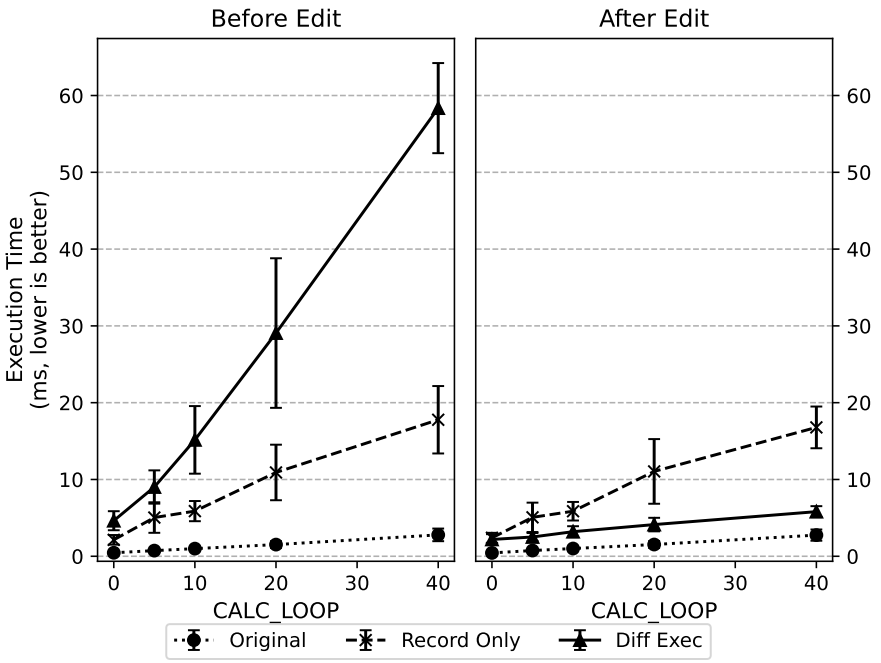

図 6.5: 連結リストを用いた場合の実行時間（CHANGED = 100）

第7章　関連研究

7.1　漸増計算

漸増計算（Incremental Computation）とは，プログラムに入力される値が更新されたときに，前回実行時の記録などを用いることによって，効率よくプログラムを実行する手法のことである [14]。

本論文の提案する差分実行方式は，以下の 2 点において漸増計算手法であるといえる。

まず 1 つ目に，ゲスト言語上のプログラムが受け取る入力についての漸増計算である。本論文においてはゲスト言語上のプログラムの入力について考慮していないため，入力値はプログラムの先頭にハードコーディングされると考えることにする。このとき，入力値が格納された変数に対して汚染フラグが付与され，その後入力変更前の実行時情報を元にプログラムを差分実行することになる。これにより，入力の更新によって影響が発生する箇所のみを再計算し，そうでない箇所については実行の省略を行う。したがって差分実行は，ゲスト言語上のプログラムが受け取る入力についての漸増計算手法として考えることができる。

2 つ目に，インタプリタが受け取る入力についての漸増計算である。この場合の入力とはインタプリタの対象言語で記述されたプログラムとなり，したがって入力の差分とはすなわち編集によるものである。差分実行方式を導入したインタプリタは，編集前のプログラムの実行時情報を用いて編集後のプログラムを効率よく実行する。すなわち，プログラム編集の影響を受ける箇所のみを再計算し，そうでない箇所については実行の省略を行う。したがって差分実行は，インタプリタが受け取る入力についての漸増計算手法としても考えることができる。

7.2　Glitch

McDirmid らは，詳細な実行順序をプログラマが定めるのではなく実行環境に委任するプログラム実現手法を提案し，その一例として Glitch と呼ばれるリアクティブプログラミングに似た実行環境を実現した [11]。Glitch を用いた開発では，プログラマはいくつかのプログラム片を記述

する。これを実行すると，Glitch はプログラム片を任意の順番で実行する。その途中において，ある一片 A の実行によってすでに実行された一片 B の実行に影響が及ぶ場合，すなわち A を実行する前の B の実行内容と A を実行した後の B の実行内容に差異が生じる場合，再度 A を実行し直す。このようにして，複数のプログラム片の実行を不動点に到達するまで繰り返す。

Glitch では，あるプログラム片の実行が他のプログラム片の実行に影響を及ぼしうるか判定するために，プログラム片実行中に発生する変数の読み書きを記録しておく。すなわち，あるプログラム片の実行中にある変数への代入が発生した際に，その変数の値を用いて実行を行ったプログラム片を収集し，そのプログラム片を再実行することによって，全てのプログラム片で矛盾がない状態まで実行する。この読み書きを用いることで，例えばイベントの処理などによって変数の一部が更新されたときに，それによって再計算の必要があるプログラム片を最小に留めることができる。また，プログラム片に編集があった際も同様に，その編集によって影響を受ける最小のプログラム片集合を取ることができ，したがってプログラム編集に対して効率的に再実行を行うことができる。

Glitch と本論文の提案手法との差異は，それぞれの手法が対象とするプログラムのスタイルと，対応する再実行の原因にある。Glitch ではプログラムの実行順序を明示しないプログラミングスタイルによって記述されたプログラムを対象としているが，本論文では現在一般的である逐次的に記述されたプログラムを対象にとる。また，Glitch ではプログラム編集による実行内容の変化だけでなく，プログラム実行中に発生する再計算に対しても適用可能であるのに対して，本論文の手法はプログラム編集による実行内容の差分に焦点を当てている。

7.3　Dynamic Taint Analysis

Dynamic Taint Analysis とは，プログラム実行中に信頼できない情報源から入力された値を**汚染**された値として記録しておき，汚染された値の拡散，即ち汚染された値を用いて算出された値を追跡し，汚染された値の不正な利用を検知する動的なプログラム解析手法である [12]。この手法は，主にソフトウェアへの未知の攻撃を動的に検出するために用いられる。

本論文における汚染フラグは，当該手法における汚染された値にほぼ対応する。すなわち，信頼できない情報源から入力された値を，編集によって変更された可能性がある値と読み替えることにより，その汚染された値の拡散を追跡しているという点で共通が見出だせる。一方で，汚染された値の情報をどのように用いるかについては差異がある。

第8章　まとめと今後の課題

本研究では，データ構造に特化したライブプログラミング環境において，プログラム編集によって実行時情報の再収集が必要となった際に，編集の差分と前回実行時に収集した特別な実行時情報を活用し実行時情報の再収集を効率良く行う**差分実行方式**を提案した。また，差分実行方式が求める履歴情報や，差分実行方式の詳細な実現方法を述べ，実際に SimpleLanguage 言語定義に差分実行方式を組み込み Kanon 仮想機械を構築した。評価においては，編集による影響範囲が小さい場合に効率良く実行時情報を収集できることを確認した。

今後の課題を 2 点挙げる。

まず 1 つ目に差分実行方式の形式化である。本論文では差分実行方式を実現するための具体的なアルゴリズムなどを示したが，実際にこれが適切に定義されているかについては保証されていない。したがって形式化を行い，差分実行を行う場合と通常の実行を行う場合とで得られる実行情報が等価であることを示す必要がある。また，より大規模な言語に対して差分実行方式を組み込む際に，自明でない実行時情報が必要となる可能性もある。これを検証するためにも，差分実行方式の形式化は必須であると考えられる。なお，すでに差分実行の形式化の試みがある [5]。

2 つ目に実現のさらなる効率化である。今回履歴情報の実現における定義は非常に簡単なものであった。しかし，評価で示したとおり編集前の初回実行時における実行時間は非常に長く，また編集の影響範囲が大きくなった際にも実行時間が悪化してしまう。これらの問題の解決策として，実行文脈から時刻への変換や，再実行における実行戦略の決定方法など，差分実行方式にて核となるいくつかの処理で最適化の余地があると考えられる。これらの処理を最適化することにより，より多くのケースで差分実行が有効になることを目指す。

参考文献

[1] Aaron, S. and Blackwell, A. F.: From Sonic Pi to Overtone: Creative Musical Experiences with Domain-specific and Functional Languages, *Proceedings of the First ACM SIGPLAN Workshop on Functional Art, Music, Modeling & Design*, FARM '13, New York, NY, USA, ACM, pp. 35–46 (2013).

[2] Archambault, D. and Purchase, H. C.: The mental map and memorability in dynamic graphs, *2012 IEEE Pacific Visualization Symposium*, pp. 89–96 (2012).

[3] Bolz, C. F., Cuni, A., Fijalkowski, M. and Rigo, A.: Tracing the Meta-Level: PyPy's Tracing JIT Compiler, *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, New York, NY, USA, Association for Computing Machinery, p. 18–25 (2009).

[4] Evans, T. G. and Darley, D. L.: On-Line Debugging Techniques: A Survey, *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), New York, NY, USA, Association for Computing Machinery, p. 37–50 (1966).

[5] Furudono, N.: プログラム差分実行方式の形式化, Master's thesis, Tokyo Institute of Techonology (2023).

[6] Hofer, C., Denker, M. and Ducasse, S.: Design and Implementation of a Backward-In-Time Debugger, *NODe 2006*, Proceedings of NODe 2006, Vol. P-88, Erfurt, Germany, GI, pp. 17–32 (2006).

[7] Kato, J., McDirmid, S. and Cao, X.: DejaVu: Integrated Support for Developing Interactive Camera-Based Programs, *Proceedings of the 25th Annual ACM symposium on User Interface Software and Technology*, UIST '12, New York, NY, USA, ACM, pp. 189–196 (2012).

[8] Lewis, B.: Debugging Backwards in Time, *Proceedings of the Fifth International Workshop on Automated Debugging* (Ronsse, M. and Bosschere, K. D.(eds.)), AADEBUG 2003 (2003).

[9] Lienhard, A., Gîrba, T. and Nierstrasz, O.: Practical Object-Oriented Back-in-Time Debugging, *ECOOP 2008 – Object-Oriented Programming* (Vitek, J.(ed.)), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 592–615 (2008).

[10] Maloney, J., Resnick, M., Rusk, N., Silverman, B. and Eastmond, E.: The Scratch Programming Language and Environment, *ACM Trans. Comput. Educ.*, Vol. 10, No. 4 (2010).

[11] McDirmid, S. and Edwards, J.: Programming with Managed Time, *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, New York, NY, USA, Association for Computing Machinery, p. 1–10 (2014).

[12] Newsome, J. and Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, *Proc. 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, pp. 123–130 (2005).

[13] 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, p. 72–87 (2018).

[14] Pugh, W. and Teitelbaum, T.: Incremental Computation via Function Caching, *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, New York, NY, USA, Association for Computing Machinery, p. 315–328 (1989).

[15] Takahashi, S., Izawa, Y., Masuhara, H. and Cong, Y.: An Approach to Collecting Object Graphs for Data-structure Live Programming Based on a Language Implementation Framework, *Journal of Information Processing*, Vol. 30, pp. 451–463 (2022).

[16] Tanimoto, S. L.: A Perspective on the Evolution of Live Programming, *Proceedings of the 1st International Workshop on Live Programming*, LIVE '13, IEEE Press, p. 31–34 (2013).

[17] Würthinger, T., Wimmer, C., Humer, C., Wöß, A., Stadler, L., Seaton, C., Duboscq, G., Simon, D. and Grimmer, M.: Practical Partial Evaluation for High-Performance Dynamic Language Runtimes, *SIGPLAN Not.*, Vol. 52, No. 6, p. 662–676 (2017).

[18] Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D. and Wimmer, C.: Self-Optimizing AST Interpreters, *SIGPLAN Not.*, Vol. 48, No. 2, p. 73–82 (2012).