令和２年度 学士論文

データ構造ライブプログラミング環境の汎言語的な構築手法

東京工業大学 情報理工学院 数理・計算科学系
学籍番号 17B09137

高橋 修祐

指導教員

増原 英彦 教授

令和３年２月５日

概要

ライブプログラミング環境は，編集中のプログラムを継続的に実行し，プログラムの変更を即座にプログラマへフィードバックする機構である。中でも Kanon はオブジェクトの参照関係をプログラマが把握しやすい形で描画するデータ構造特化ライブプログラミング環境である。これまでのライブプログラミング環境は，特定のプログラミング言語に特化して実装されてきた。しかし Kanon の有用性を高めるために，複数言語に対応した処理系として実現すべきである。本研究ではデータ構造特化ライブプログラミング環境の汎言語化をメタコンパイラフレームワークである Graal/Truffle を用いて実装する手法を提案する。

謝辞

本研究を進めるにあたって多くのアドバイスやご指導をしてくださった増原英彦教授，叢悠悠助教に心より感謝申し上げます。

また，日頃から多くの助言をしていただいた増原研究室のみなさまにも感謝致します。特に本研究に際し背景知識等をご教示頂いた伊澤侑祐さんには感謝の念が絶えません。

目 次

第 1 章　序論 **5**

第 2 章　背景 **7**
- 2.1 ライブプログラミング環境 7
- 2.2 Kanon 7
  - 2.2.1 Kanon が収集する実行時情報 7
  - 2.2.2 Kanon の実行手順 10
- 2.3 Graal/Truffle 11
  - 2.3.1 GraalJS 11
  - 2.3.2 Instrumentation API 12
- 2.4 LSP 13

第 3 章　既存実装の問題点と提案手法 **14**
- 3.1 多言語化及び多開発環境化のための既存実装とその問題点 14
- 3.2 提案手法:メタコンパイラフレームワーク及び LSP を用いた実装 15

第 4 章　実装 **16**
- 4.1 オブジェクト操作の記録 16
- 4.2 対応するソースコード上の位置の取得 17
- 4.3 スタックトレースの取得 18
- 4.4 組み込みオブジェクトの排除 19

第 5 章　関連研究 **20**

第 6 章　結論 **22**

図 目 次

2.1 Kanon 8
2.2 各時点におけるオブジェクトの参照関係 9
2.3 Kanon の実行手順 10

5.1 VSCode 上で動作する Babylonian Programming System . 20
5.2 OnlinePythonTutor 21

第1章　序論

ライブプログラミング環境は，編集中のプログラムを継続的に実行し，プログラムの変更を即座に反映する機構である。これにより，環境利用者はプログラムの編集をしてからすぐにそれによる変更を確認することができる [7]。特にデータ構造に特化したライブプログラミング環境は，プログラムの実行中における各オブジェクトの参照関係を，視覚的に関係が把握しやすいものへと変換し表示する。Kanon[6] は，データ構造に特化したライブプログラミング環境のひとつである。Kanon によって，プログラマは自身の書いたプログラムにおいてオブジェクトがどのように作成・変更されるかを容易に確認することができる。

既存のライブプログラミング環境は，特定の 1 つのプログラミング言語に特化して実装されてきた。例えば Kanon は JavaScript のプログラムのみに対応した環境である。一方で，実用向けのライブプログラミング環境は複数の言語への対応が可能である場合がある。例えば，先述のデータ構造に特化したライブプログラミング環境は，データ構造がプログラミングをする際に言語によらず用いられる概念であるため，多言語対応が可能である。現在では Java や Python 等多数の言語が実用されていることを考えると，このような環境はその普及を図るために複数の言語に対応していることが望ましい。

またプログラムを実装する際に用いられる開発環境への対応も考えると，多言語化に合わせて，多開発環境化も実現するべきである。プログラミングを補助するようなライブプログラミング環境は，独自のエディタ等を提供せずに既存のエディタを利用したほうが，その環境に対応する他のプラグイン等が提供する機能も同時に使うことができるため好ましい。しかし今日では多数の開発環境が存在する。プログラマに多様な選択肢を与えるためにも，多言語化に加えて，特定の言語に特化した開発環境それぞれに対応すべきである。

多言語化を達成するための方法としてまず考えられるのが，動的解析を行う部分を対応させる各言語ごとに分け，開発者に情報を提示する部分を共通化するという方法である。この方法では，各言語に特化したランタイムを用いて実装できるため性能の調整が容易という利点を持つ。しかし，各言語ごとに環境の実装をする必要があり，実装にかかる手間が莫大に

なってしまうという問題がある。

本研究では (1) 汎言語及び (2) 汎開発環境に対応したデータ構造特化ライブプログラミング環境を構築する。手法として，(1) メタコンパイラフレームワーク上で環境を構築し，(2)Language Server Protocol (LSP) を用いて開発環境と通信することを提案する。まず (1) によって，採用したメタコンパイラフレームワークが対応するすべての言語において環境を利用することが可能となる。また (2) によって，複数の開発環境への対応を容易にすることが可能となる。

以降における本稿の構成は次の通りである。第 2 章でライブプログラミング環境とその一種である Kanon，メタコンパイラフレームワークである GraalVM 及び Truffle 及び LSP について簡単に説明する。第 3 章でライブプログラミング環境の多言語・多開発環境対応の必要性とそれらを実際に実装する上での問題点について挙げ，またそれらの問題を解決するため，メタコンパイラフレームワークと LSP を用いたライブプログラミング環境の実装を提案する。第 4 章でこれらを用いて実装される Poly^{2}Kanon の解析部の実装方針について述べる。第 5 章で関連研究を挙げ，第 6 章でまとめと今後の展望について述べる。

第2章　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とは，入力されたプログラムが編集されたと同時にそのプログラムを動的あるいは静的に解析し，そこから得られた情報を即座にプログラマに表示するシステムである。用途として，プログラミング教育 [3] や，あるいは芸術への応用 [1] 等が挙げられる。この他の用途として，特にコーディング等の用途で用いられるライブプログラミング環境は，プログラムを編集し，変更されたプログラムを実行してその動作を確認して，プログラムの記述を再開するというサイクルを一つにまとめようとする。これによってプログラマは，修正したプログラムを逐一ターミナル等で実行せずとも自らが記述したプログラムの妥当性を確認することができる。

2.2　Kanon

Kanon はデータ構造に特化した JavaScript 向けライブプログラミング環境の一つである。ブラウザ上で動作する Kanon の画面を図 2.1 に示す。Kanon では，Kanon のエディタ上 (図左側) で開発中のプログラムに変更があるたびに，そのプログラムを実行し，実行中に生成されたオブジェクト間の参照関係をグラフにして描画する (図右上)。開発者は表示されたグラフからプログラムが意図したとおりに実装されているかを確認し，連続的にプログラムの開発を進めることが可能となる。

2.2.1　Kanon が収集する実行時情報

Kanon はオブジェクト間の参照関係をグラフとして描画するために，与えられたプログラムを動的に解析する。具体的には次の 2 点を動的解析によって記録する。

オブジェクトの参照関係

オブジェクトの参照関係を得るには，次の 3 つの値が必要となる。

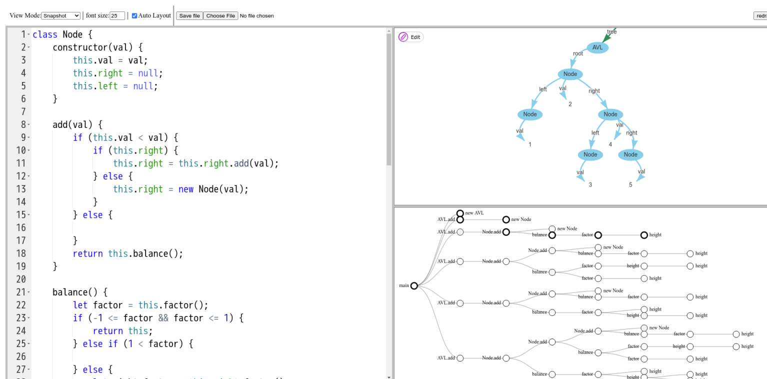

図 2.1: Kanon

1. ローカル変数が持つ値
2. グローバル変数が持つ値
3. オブジェクトや配列の各フィールドが持つ値

例えばソースコード 2.1 の 2 行目の文が実行された後 (ソースコード中▲の地点) を考える。1 つ目のローカル変数について，`object` というローカル変数はあるオブジェクト (ここではオブジェクト A と呼ぶ) を持つ。2 つ目のグローバル変数について，`x` というグローバル変数は数値 `1` を持つ。3 つ目の各フィールドが持つ値について，オブジェクト A が `f` というフィールドを持ち，その `f` の持つ値は数値 `1` である。

ソースコード 2.1: JavaScript プログラムの例

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

また，Kanon においてはソースコード中のどの箇所において参照関係が発生し，変更され，または消滅したのかも記録する。これを利用し，ソースコード上のある一点を指したときに，その点に到達するまでに構成された参照関係の提示を行っている。従って Kanon では，プログラム終了時における 3 つの値を記録するのではなく，プログラム中の各文が実行される前後でこれを記録している。詳細には，各文が実行される前後で，その

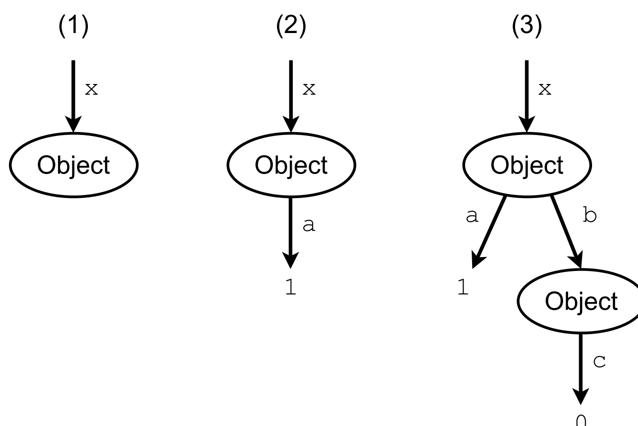

図 2.2: 各時点におけるオブジェクトの参照関係

文から見ることができる変数及びオブジェクトのすべてを探索して記録している。

例えばソースコード 2.2 を実行した場合を考える。このソースコードは (1) から (3) までの 3 つの文から構成されており，それぞれを実行した直後の参照関係は図 2.2 に示したようになる。これらの参照関係をそれぞれ保存するために，それぞれの文が実行される前後で 3 つの値を保存している。

ソースコード 2.2: JavaScript プログラムの例

```
let x = {}; //(1)
x.a = 1; //(2)
x.b = {c: 0}; //(3)
```

ソースコード編集の前後における文脈の維持

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，生成されたグラフを触ることが可能である。即ちプログラマは，プログラムの動的な解析を経て生成されたグラフを拡大・縮小したり，あるいはグラフ中のあるノードの位置を変更したりすることができる。そして，ソースコードの変更によってグラフを再描画する際に，プログラマによるグラフの変更を維持する。

この機能を実現するために，Kanon は同じソースコード上の箇所から生成されたオブジェクトに対して，それぞれを区別し，またソースコードが変更された際に，変更前のソースコードのある箇所が変更後どこに対応

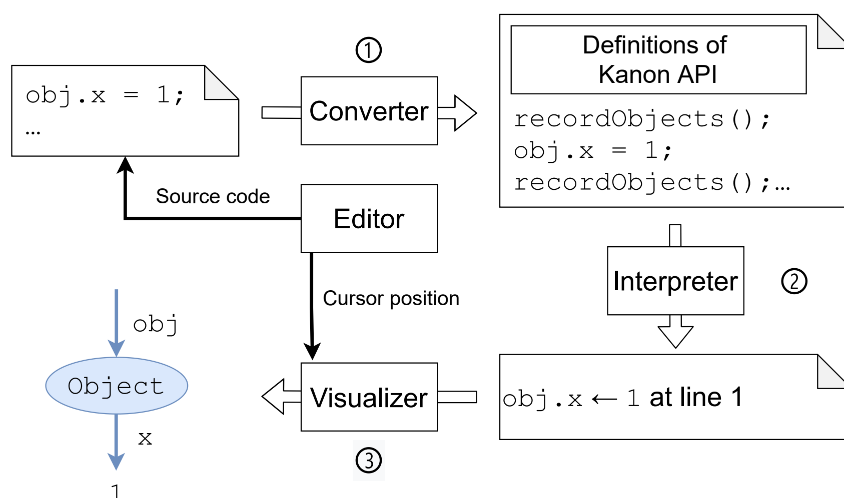

図 2.3: Kanon の実行手順

しているのかについても記録している。具体的には，実行時のオブジェクトの生成あるいは参照関係の変更が行われた際に，そのオブジェクト操作が発生したソースコード上の位置に加えて，変更発生時のスタックトレースを記録する。これにより，for 文等の反復を行う制御構造内や同一の関数が複数回呼び出され，オブジェクトが複数生成されたときに，それらのオブジェクトを区別することが可能となる。

2.2.2　Kanon の実行手順

Kanon が提供するエディタにてプログラムが変更されてから，そのプログラムの変更を反映したグラフを描画するまでの実行手順を図 2.3 に示す。

1. 変更されたプログラム全体に対して構文解析を行う。その後，2.2.1 節で示した 2 点を収集するために，与えられたプログラムを変換する。具体的には，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。チェックポイントが実行されると，そのとき変数から辿ることのできるすべてのオブジェクトや文脈等が記録される。また，この記録を行うために必要となる関数や変数等も併せて挿入される。

2. 変換したプログラムを `eval` 関数を用いて実行し，チェックポイントで得られる情報を収集する。

3. エディタ上のカーソルの位置から，対応するチェックポイントを決定し，その時点におけるオブジェクトの参照関係をグラフ化して表示する。

2.3　Graal/Truffle

メタコンパイラフレームワークとは，ある言語のインタプリタを定義することによって，その言語の JIT コンパイラ付き仮想機械を生成することができる処理系をいう。その一例として Graal/Truffle[8] がある。

言語実装者は，Truffle と呼ばれる GraalVM が提供する言語実装用のフレームワークを用いながら，抽象構文木を基にしたインタプリタを実装する。これにより，その言語の高性能な仮想機械を生成することができる。GraalVM は，インタプリタを自身の上で実行させる際に Just-in-Time コンパイルを行うため，高速にプログラムを実行することができる。また，GraalVM は JVM の拡張であるため，言語実装者がインタプリタを実装する際には，JVM が提供する巨大で安定したコンポーネント群，例えばガーベッジコレクションやスレッディング等を利用することができる。これにより高性能な仮想機械を容易に実装することができる。

2.3.1　GraalJS

GraalJS は GraalVM 上で動作する JavaScript インタプリタである。本研究の実装例として JavaScript を取り扱うため，本節では GraalJS を例にとって GraalVM 及び Truffle を用いたインタプリタの実装を説明する。

まず抽象構文木のノードに当たる部分をクラスとして定義する。このとき Truffle が提供するクラス `Node` を継承する必要がある。`Node` クラスは当該ノードがソースコード上のどこに対応するかを返却する `getSourceSection` メソッドを提供する。

これに加えて GraalJS では，ノードの基底クラスとして `Node` を継承させたクラス `JavaScriptNode` を用いる。このクラスにはそのノードを実行させるための抽象メソッド `execute` メソッドが定義されている。`execute` メソッドは，子ノードの `execute` メソッドを実行し，その返り値を用いることによって実装される。

Graal 及び Truffle においてオブジェクトは次のように実装される。まずデータを実際に持つのは抽象クラス `DynamicObject` を継承したクラスである。これらのクラスはデータは持つものの，例えばオブジェクトにおけるキーと値等，識別子とデータがどのように紐付いているかという情報を直接保持しない。これらの情報を保持するために，`DynamicObject` は

Shape のインスタンスをメンバに持ち，かつ Shape は Property のインスタンスをメンバに持つ。Property が識別子と DynamicObject 内の位置を結びつけており，Shape はオブジェクトの構造に合わせてその分だけ Property を保持する。従って，あるオブジェクトに新しいフィールドを定義して，そこに値を代入する際には DynamicObject と Shape が更新され，かつ新しい Property のインスタンスが生成されることとなる。また，既存のキーに結びついた値を更新する場合には，当該 DynamicObject のインスタンスと，そのオブジェクト中の位置を表す Property のインスタンスが必要となる。

2.3.2　Instrumentation API

GraalVM 上で汎言語的に動作するツールを開発するために，Truffle は Instrumentation API を提供する。これを利用することで，インタプリタがプログラムを実行する際の挙動を観測したり，あるいは新たな挙動を差し込むことができる。ただし，Graal 及び Truffle 上で構築されたすべてのインタプリタで無条件に使える訳ではなく，インタプリタ側が Instrumentation API が利用できるように実装をする必要がある。

Instrumentation API が提供する機能の一つに，特定のノードが実行される前後における任意のプログラムの実行がある。インタプリタがこの機能を提供するには，抽象構文木のノードを表すクラスを実装する際に，クラス Node を継承するのに加えて，インタフェース InstrumentableNode を実装する必要がある。InstrumentableNode はメソッド createWrapper を提供する。このメソッドを呼び出すと，当該ノードを包んだノードを返却する。即ち，返却されたノードがもつ execute メソッドを呼び出すと，その中で元のノードの execute メソッドが呼び出される。更に，元のノードの execute メソッドを実行する前，実行した後，あるいは元のノードの execute メソッド実行時に例外が発生した場合にツール側が指定するプログラムを実行することができる。

ツール側ではどのノードに対して，どのタイミングで，どのようなプログラムを挿入するかを指定する。まず，クラス ExecutionEventNode を継承したクラスと，インタフェース ExecutionEventNodeFactory を実装したクラスをそれぞれ定義する。この 2 つによって，どのプログラムをどのタイミングで実行するのかを指定することができる。次に，インタプリタがプログラムを実行する前に，インタプリタがもつ環境にインタフェース ExecutionEventNodeFactory を実装したクラスを追加する。この追加の際に，併せてどのノードに対してプログラムを挿入したいのかを指定することができる。即ち，一部のノードの実行前後に独自のプログラムを挿入したい場合に，ノードの指定をすることができる。指定条件としては，

プログラム名やソースコード上の位置等で指定することができる。また，`InstrumentableNode` はメソッド `hasTag` をもち，このメソッドをオーバライドすることによってノードの種類を示すことが可能である。よってタグを用いてノードの条件を設定することもできる。

2.4 LSP

LSP (Language Server Protocol)[4] は，自動補完や定義へのジャンプ等の機能を提供する言語サーバとエディタとの間の通信内容について定めた規約である。いままではプログラムを解析してその結果を既存のエディタに出力するためには，各エディタの提供する API によって実装する必要があった。従って，他のエディタに機能を移植するためには，エディタに依存した箇所について新たに書き直す必要があった。これに対し，エディタの提供する API によって実装する箇所を LSP を用いて実装することにより，エディタを跨いだ機能の移植が容易となる。

第3章　既存実装の問題点と提案手法

3.1　多言語化及び多開発環境化のための既存実装とその問題点

既存のライブプログラミング環境の多くは対象とする言語が一つに定まっていた。これは，各環境の達成しようとする目的に合わせて独自の言語を構築し，その言語に対応するように環境を実装していること等が理由として挙げられる。一方で，プログラミングに共通した要素に特化したライブプログラミング環境については，言語を限定することなく環境を構築することが可能である。例えば，データ構造はプログラムを実装する上で言語を問わず必須となる要素であり，これに特化したライブプログラミング環境は多言語に対して提供することが可能である。

しかし，プログラム実行時のデータ構造の解析を多言語対応することは技術的な困難が伴う。理由として，実行時のデータ構造を取得するには，そのプログラムが実行中に持つスタックやヒープを取得する必要がある。従ってプログラムをうまく変換してそれらの情報が取れるようなプログラムを差し込むか，あるいは言語のランタイムに解析をするための改変を行わなければならない。複数の言語のプログラムを統一的に変換することは困難であり，また一般的なランタイムが対応する言語は 1 つであるから，そのままでは多言語対応することはできない。

これを回避するために，解析部を各言語ごとに実装し，それ以外の部分を共通化するという方法がある。この実装方針を採用した一例として OnlinePythonTutor[2] が挙げられる。OnlinePythonTutor は当初 Python のみに対応したライブプログラミング環境であったが，現在では Python に加えて Java や C，C++，JavaScript，Ruby に対応している。OnlinePython-Tutor の実装は，与えられたプログラムを動的に解析するバックエンドと，エディタの提供及びバックエンドからの情報をプログラマに提示するフロントエンドの 2 つに分かれている。フロントエンドに関しては各言語で共通化されている一方，バックエンドにおいては対応する各言語によってそれぞれ実装が異なっている。このような実装方針においては，各言語ごとに最適な実装を選択することが可能であるため，性能の調整が容易になるという利点がある。しかし，新しい言語に対応させるときには，その言語

に合わせた実装を再度検討する必要がある。また，環境そのものの仕様を変更する際には，各言語の実装すべてに修正を加えなければならない。これらの理由から，環境実装者の負担が大きくなってしまう。

　また，多言語に合わせて複数の開発環境に対応することも必要である。しかし，既存の開発環境を拡張するには，それぞれの開発環境が提供する API 等を用いて実装する必要があり，これもまた環境実装者の負担となる。

3.2　提案手法:メタコンパイラフレームワーク及び LSP を用いた実装

　環境を汎言語及び汎開発環境的にかつ容易に実装するために，本論文ではメタコンパイラフレームワークと LSP を用いてライブプログラミング環境を実装することを提案する。具体的には，メタコンパイラフレームワークが提供するオブジェクトの生成や変更をするための API に記録用の記述を付属させることにより，インタプリタが行うすべてのオブジェクトを操作する記録を取ることが可能となる。これにより，当該フレームワーク上で実装されたインタプリタについては，すべてのオブジェクト操作の記録が可能となり，汎言語的な環境の実装を達成することができる。また，LSP を用いることにより，開発環境ごとに異なる拡張方法の差をできるだけ小さくすることができ，汎開発環境的な環境の実装も達成することができる。

第4章　実装

第3章で提案した手法の妥当性を検証するため，Kanon を汎言語化させた Poly^{2}Kanon を実装中である。本章では特に Poly^{2}Kanon の解析部分の実装を述べる。まず，今回メタコンパイラフレームワークとして Graal 及び Truffle を選択した。また汎言語化を達成するに当たり，インタプリタ側においてもある程度の実装の変更が必要となる。今回は GraalVM 上で動作する JavaScript のインタプリタである GraalJS に改変を加えることとして，以降説明する。他の言語のインタプリタに関しても，以下と同様の方針で実装することができる。

Poly^{2}Kanon の解析部分の実装は大きく3点に分けられる。(1) まずオブジェクトの生成や更新に関わるフレームワークの API の呼び出しがあったときに，環境がもつオブジェクトの解析用の関数を同時に呼び出す。これによって，プログラム実行時のオブジェクト操作の全履歴を取ることが可能になる。

(2) これに加えて，オブジェクトの操作があったとき，ソースコード上のどの箇所を実行していたのかを記録する。これによってエディタ上のカーソルを移動したときに，そのカーソル付近が実行されたときのオブジェクトの参照関係のグラフを生成することが可能となる。

(3) 更に，オブジェクトの操作があったときのスタックトレースを記録する。これによりソースコード上の同じ箇所が複数回実行され，複数のオブジェクトが操作された場合にも，グラフ上のオブジェクトがどのタイミングで生成されたのかすべて識別することができる。

4.1　オブジェクト操作の記録

Graal および Truffle で実装されたインタプリタは，ソースコードを抽象構文木に変換し，そのノードを表すクラス `Node` のメソッド `execute` を呼び出すことによってプログラムを実行する。従って，オブジェクトを操作しうるノードについては，メソッド `exeucte` 内でオブジェクトを実際に操作することとなる。また，このメソッド `execute` 内では，オブジェクトを操作するために，操作するオブジェクト，オブジェクト内の変更される部分 (例えば識別子等)，及び新しい値が取得できると期待できる。従っ

て，これら 3 つの値により，オブジェクト操作の記録を取ることができる。

一方で GraalJS においては，最適化を図るためにオブジェクト操作が抽象化されている。これにより，オブジェクトの操作はメソッド execute 内で直ちに行われない。例えば，オブジェクト操作をするノードに対応する GraalJSNode の子クラスは，クラス PropertySetNode のインスタンスをメンバに持つ。またクラス PropertySetNode は抽象クラス SetCacheNode のインスタンスをメンバに持つ。SetCacheNode は操作するオブジェクトや，代入される値の型等に合わせて最適なものが選択される。SetCacheNode の子クラスでは，クラス Propety のインスタンス等をメンバに持ち，実際にオブジェクトが操作される。

この抽象化により，オブジェクトを実際に操作するのは SetCacheNode などのソースコードから直ちに生成されないようなクラスとなる。即ち，ソースコードから抽象構文木を経由して生成された GraalJSNode の子クラスの execute メソッドでは，オブジェクト内の変更される部分を取得することができない。従って，Property のインスタンスが実際に扱われるような箇所においてオブジェクト操作を記録することとなる。前の例では，SetCacheNode の子クラスのメソッド execute の中にオブジェクト操作記録用の記述をすることとなる。

具体的な実装は次の通りである。まずオブジェクト操作に対するリスナ用のインタフェースと，インタプリタ内でオブジェクト操作を受け，登録されたリスナにそれを通知するクラス ObjectTracker を定義する。次に，インタプリタが実行時に持つ環境にメンバとして ObjectTracker を追加し，環境の初期化処理としてインタプリタに ObjectTracker を渡すようにする。最後に，オブジェクトの操作を行うすべての箇所に，ObjectTracker 内にあるオブジェクト操作があったことを通知するメソッドを呼び出すプログラムを挿入する。

4.2　対応するソースコード上の位置の取得

前節で述べた通りソースコード上のオブジェクト操作に対応する JavaScriptNode の子クラスと，実際にオブジェクトが操作されるクラスは別である。前者については抽象構文木から直ちに生成されるから，ソースコード上の位置に関する情報を有する。一方で，後者はそのような情報を保有しない。従って，後者のクラス内のオブジェクトを記録するような箇所において，同時にソースコード上の位置を記録することは不可能である。

これを解消するために，Truffle が提供する Instrumentation API を利用し，オブジェクト操作がなされたあとに，最初に実行が完了した execute

メソッドを持つノードがオブジェクト操作を行ったノードであるとみなす。このようにして，オブジェクト操作とその原因となったソースコード上の箇所を対応付ける。この方針は，GraalJS のように，ソースコードから直ちに生成されるノードと，実際にオブジェクト操作を行う関数が異なっている場合に限らず，ノードの execute メソッド内で直接オブジェクトを操作するようなインタプリタに対しても有効な方針である。

具体的な実装としては，ExecutionEventNodeFactory 及び ExecutionEventNode を用いて，すべてのノード実行の完了を記録できるようにする。前節の実装によってオブジェクト操作が発生したと確認できたとき，それ以降最初に完了したノードのソースコード上の位置を取得し，オブジェクト操作の記録にそれを書き加える。尚，Node の getSourceSection が返す値は null である可能性があるので，オブジェクト操作後の真に最初のノードを推定するのではなく，初めて null でない値が返されたノードのソースコード上の位置を記録する。

4.3　スタックトレースの取得

スタックトレースを取得するために例外を利用する。Java における例外を表す基底クラス Throwable には，メソッド getStackTrace が定義されており，通常のスタックトレースを取得するにはこれを呼び出せば十分である。一方で，GraalJS が実行中のプログラムに関してスローする例外はすべてクラス GraalJSException を基底とする。即ち，例えばソースコード 4.1 のようなプログラムにおいては，一般的な Java のプログラムと同様 NullPointerException がスローされるのではなく，GraalJSException の子クラスのインスタンスがスローされることとなる。このクラスは当然 Throwable を継承し，かつメソッド getStackTrace をオーバーライドする。これにより，GraalJSException のメソッド getStackTrace を呼び出した場合に，インタプリタのスタックトレースが返却される代わりに，インタプリタ上で動作しているプログラムのスタックトレースが返却される。

従って，オブジェクトの操作が発生したときに，GraalJSException のインスタンスを生成して，その getStackTrace メソッドを呼び出すことによってスタックトレースを取得することができる。

今回の実装では，例外を生成する代わりに，GraalJSException のメソッド getStackTrace 内で呼び出されるスタックトレース取得用のメソッドを直接呼び出してスタックトレースを取得している。このスタックトレースの記録は，前小節におけるオブジェクト操作が行われてから最初に完了したノードのソースコード上の位置を記録するときに併せて行う。

ソースコード 4.1: `GraalJSException` を投げるソースコードの例

```
1 null.field = 1;
```

4.4　組み込みオブジェクトの排除

JavaScript を含む多くの言語では組み込みのオブジェクトが存在する。しかし，これらのオブジェクトはグラフに描画する必要がないため，これを取り除く必要がある。組み込みのオブジェクトは通常ソースコードの最初の文を実行する前に利用可能でなければならないから，組み込みのオブジェクトはプログラム実行前に登録されると考えられる。従って，最初のノードが実行される前に発生したオブジェクト操作はすべて組み込みのオブジェクトであるとみなして，これを無視することとする。

これを行うために，ソースコードの記録等と同様，Instrumentation API の `ExecutionEventNodeFactory` 及び `ExecutionEventNode` を用いる。プログラム中の最初のノードが実行される直前を記録し，それより前に発生したオブジェクト操作の履歴をすべて破棄する。

第5章　関連研究

同じくGraalVMとTruffle，LSPを用いた汎言語的・汎開発環境的なライブプログラミング環境の実装として，Babylonian Programming System (BPS)[5] がある。図5.1はJavaScriptのプログラムをVSCodeを介してBPSで解析している画面である。BPSは，エディタに記述した関数等について，エディタ上からその関数の引数を仮に指定し，即座にその返り値を取得する等，コーディングとデバッグをエディタ上で同時に行えるようにした環境である。実装上の違いとして，BPSはGraalVM上で扱える情報で動的解析をするのに加え，現状のGraalVMでは解析できない部分について，独自に静的な解析を行うことによって解決している。一方本研究においては，GraalVMの内部の実装を変更し，すべての解析をGraalVMで行えるよう試みている。

```
1  // <Example :name="ten" n="16" />  987
2  // <Example :name="eight" n="8" />  21
3  function fibonacci(n) {
4      let x = 0;
5      let y = 1;
6      for (let index = 0; index < n; index++) {
7          let z = x;
8          // <Probe />  1→1→2→3→5→8→13→21→34→55→89→144→233→377→
9          x = y;
10         y = z + y;
11     }
12     // <Assertion :example="eight" :expected="21" />  true
13     return x;
14 }
15
16 fibonacci(5);
17
```

図5.1: VSCode上で動作するBabylonian Programming System

また，既に多言語に対応したデータ構造に特化したライブプログラミング環境としてOnlinePythonTutor[2] がある。Kanonと同様ソースコードを実行して，実行中の各時点に構築されているオブジェクトの参照関係をグラフ化して表示する環境である (図5.2)。現状はPythonに加えてJavaScriptにも対応している。OnlinePythonTutorの実装は前述の通り

で，エディタやグラフ等の表示を行う部分については各言語共通とし，動的解析を行う部分に関しては言語ごとに実装を分けている。一方本研究では動的解析を行う部分においても実装を共通化することにより，実装の負担をできるだけ小さくしている。

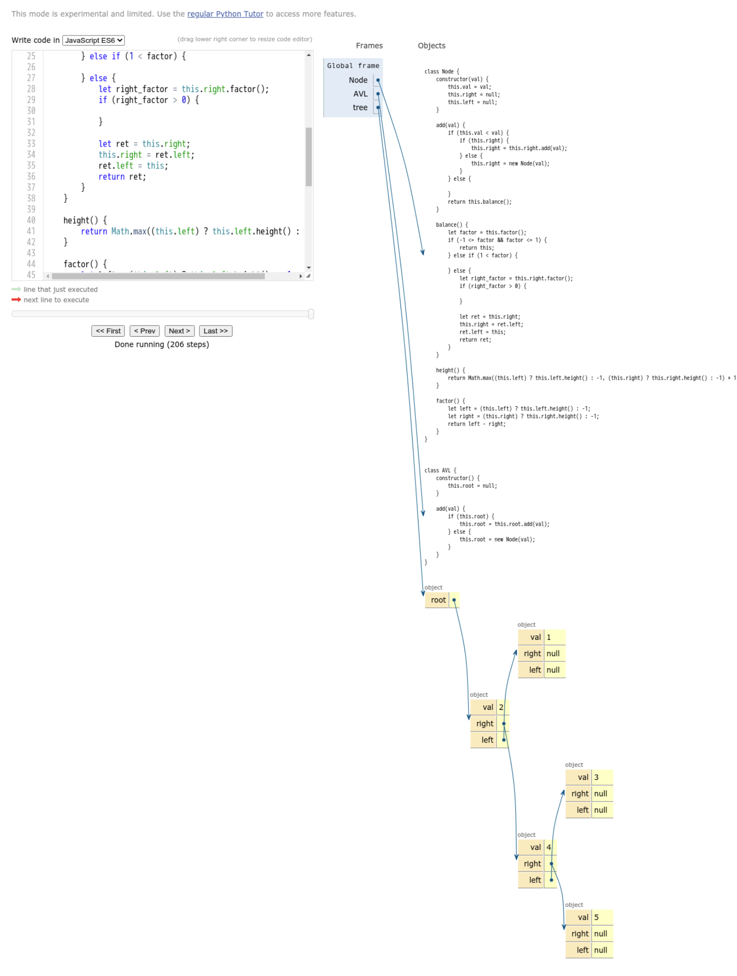

図 5.2: OnlinePythonTutor

第6章　結論

本論文では，汎言語的なライブプログラミング環境をメタコンパイラフレームワーク及びLSPを用いて構築することを提案した。また，データ構造に特化したライブプログラミング環境の解析部分について，Graal及びTruffleを用いたときの実装方針を示した。これにより，ライブプログラミング環境を少ない実装コストで汎言語化させることが可能となる。

今後の課題として，まず本実装におけるライブ性の評価を行う必要がある。以前のKanonの実装では，プログラムの大きさが大きくなるほど描画までに時間がかかっていた。その理由の一つに，動的解析をする際に，ソースコードの各文を実行するたびにその時点で辿ることができるすべてのオブジェクトを記録していたことが挙げられる。一方，今回提案した手法では，オブジェクトの生成や参照関係の変更が起こったときにその詳細を記録するようにしたため，この点が改善されていると考えられる。

次に，汎開発環境のための実装方針を定めることが必要である。本論文では，汎言語化への方針は示したが，汎開発環境への対応は触れることができなかった。現時点ではLSPを用いた汎開発環境的な環境の実現を目指しているが，現在のLSPでは，グラフ等の画像をエディタに提供することができない。これを回避するために，LSPの拡張を含めた方針の検討を行う必要がある。

そして，これらの実装方針をもとに，Poly^2Kanonの実装を完了させ，その実装のコストや性能等の評価等も行う必要がある。

参考文献

[1] Blackwell, A. and Collins, N.: The Programming Language as a Musical Instrument (2005).

[2] Guo, P. J.: Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education, *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, New York, NY, USA, Association for Computing Machinery, p. 579–584 (2013).

[3] 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).

[4] Microsoft: Official page for Language Server Protocol, `https://microsoft.github.io/language-server-protocol/`.

[5] Niephaus, F., Rein, P., Edding, J., Hering, J., König, B., Opahle, K., Scordialo, N. and Hirschfeld, R.: Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM, *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2020, New York, NY, USA, Association for Computing Machinery, p. 1–17 (2020).

[6] 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).

[7] 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).

[8] 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).