修 士 論 文 の 和 文 要 旨

研究科・専攻	大学院　情報理工学研究科　　情報・ネットワーク工学専攻　博士前期課程		
氏　　名	西谷 幸太郎	学籍番号	2031113
論 文 題 目	値への追跡子付与を活用したデバッグ機構		

要　旨

プログラムを実行すると，様々な値が生成・参照されながら計算が進む．プログラム実行時の値の情報は，そのプログラムの意味を理解するために重要であり，デバッグにおいても有益な情報を含むと考えられる．流体の流れを追跡するために使用される物質として追跡子というものがあり，これは生体内での薬品の伝播の状態やその範囲の調査に使用される．本研究では，追跡子による調査のアイデアをプログラムの領域に活用した，プログラム実行時の値の情報を取得するための新しい手法を提案する．この手法では，プログラム中の追跡子付与ポイントを通過した各値に追跡子を付与する．追跡子は，付与ポイントの場所情報や，値の生成の際にどの値が参照されたかの情報，ユーザ指定の文字列情報を含んでいる．

また，値への追跡子付与によって取得した情報をデバッグに有効活用するために，それらを適切に処理し，関心のある情報のみを抽出するためのクエリ言語を提案する．このクエリ言語では，値が持つ追跡子の調査や，ある追跡子を持つ/持たない値の集合の取得，値の集合に対する集合演算を行うためのクエリを備えている．それぞれのクエリは比較的単純な処理を行うものであるが，複数のクエリを組み合わせることでより複雑なクエリを作成することが可能である．そのため，ユーザの関心事をクエリとして柔軟に表現することができる．

値への追跡子付与による情報収集手法と，収集した情報を有効活用するためのクエリ言語の組み合わせによって，新しいプログラムデバッグ機構を実現する．この有用性を示すために，本研究では，JavaScript のプログラムを対象として本デバッグ機構によるデバッグを行うことができるツールを実装した．実装したツールを利用して，意図しない値が出力されたプログラムのデバッグのほかに，関心のない値の除外，プログラムの性質の検査などが可能であることが分かった．この結果から，比較的小規模なプログラムにおいて，本デバッグ機構が有用であることが分かった．

令和３年度 修士論文

値への追跡子付与を活用したデバッグ機構

電気通信大学 大学院情報理工学研究科
情報・ネットワーク工学専攻
2031113　西谷 幸太郎

指導教員　小宮 常康　准教授
　　久野 靖　　教授

提出日　令和 4 年 3 月 19 日

目次

図目次

表目次

ソースコード目次

第 1 章

はじめに

本研究では，値への追跡子付与によって収集した情報（追跡子情報）と，その情報を有用に活用するためのクエリ言語の組み合わせによって，新しいプログラムデバッグ機構を実現する．

プログラムを実行すると，様々な値が生成・参照されながら計算が進む．例えばソースコード 1.1 の JavaScript [2] プログラムを実行すると，1 行目の値 1 は変数 a に格納されたのち，2 行目で変数 a へのアクセスによって取り出される．また，2 行目の式 a + 2 の評価結果として生成された値 3 は，変数 b に格納されたのち，3 行目で変数 b へのアクセスによって取り出され，関数 console.log の実引数となる．このように，プログラム実行時の値の流れを理解することは，そのプログラムの意味を理解するために重要である．

ソースコード 1.1 JavaScript プログラム

```
1 let a = 1;
2 let b = a + 2;
3 console.log(b);
```

追跡子とは，流体の流れを追跡するために使用される物質のことであり，生体内での薬品の伝播の状態やその範囲の調査に使用される．本研究では，追跡子による調査のアイデアをプログラムの領域に活用した，新しい動的なデータ収集手法を提案する．この手法では，プログラム中の追跡子付与ポイントを通過した各値に追跡子を付与する．追跡子は，例えば付与ポイントの場所情報などを含んでおり，それらの情報を活用して，デバッグやプログラム解析に必要な操作を実現する．

関心のある値を直接的に特定し，その値に狙いを絞って追跡子を付けることができないこともある．しかしそのような場合でも，例えば，関心のない値に追跡子を付け，その追跡子を持たない値を探すことで関心のある値とその流れを見つけ出せることもある．そのためには，ある追跡子を持つ/持たない値の集合に対する集合演算等を施せればよい．そこで本研究では，追跡子が持つ情報をより有効に活用できるように，指定した追跡子を持つ/持たないなどの追跡子に関する条件を満たす値のみを絞り込むためのクエリ言語を提案する．この言語は集合演算をベースとしており，複数のクエリを組み合わせてより複雑なクエリを作成することが可能である．そのため，ユーザの関心事をクエリとして柔軟に

表現することができる．

本論文の構成として，第 2 章では提案する値への追跡子付与手法とクエリによるフィルタリング手法の概要について示す．第 3 章では提案する手法を活用したツールの実装について示す．第 4 章では，実装したツールを利用して，典型的なバグ入りプログラムを例としてデバッグを行う方法について示す．第 5 章では，本手法の問題点と課題を示す．第 6 章では関連研究を示し，最後に第 7 章では本研究の結論を述べる．

第 2 章

提案手法

この章では，本デバッグ機構を実現するための手法である，値への追跡子の付与と，クエリ言語の概要を示す．

2.1 値への追跡子の付与

本手法において，プログラム実行時のすべての値は，複数個の追跡子を自身に保持する能力を持つ．値は，式を評価した結果として得られるが，値への追跡子の付与もこのタイミングで行う（追跡子付与ポイント）．このとき，付与する追跡子には，式を評価した時刻（追跡子付与ポイントを通過した時刻）を暗黙的に含める．また，1 つの追跡子付与ポイントで複数の追跡子を付与することができる．そのため，例えば式 exp の評価の際に複数の追跡子 $m_1, \ldots, m_n$ を付与する場合，それらの追跡子に暗黙的に含める時刻は，式 exp を評価した時刻となり，追跡子 $m_1, \ldots, m_n$ が持つ時刻は同時刻となる．また，各値は暗黙的に ID を持つ．

本手法では，以下の 3 種類の追跡子を導入する．

- 値が通過したプログラム中の場所を記憶する追跡子
- 値の生成源の値（生成のために使用された値）を記憶する追跡子
- ユーザ指定の文字列を記憶する追跡子

2.1.1 導入する追跡子の概要

値が通過したプログラム中の場所を記憶する追跡子

説明のために，1 章で示したプログラムを再掲する（ソースコード 2.1）．

ソースコード 2.1　JavaScript プログラム（再掲）

```
1 let a = 1;
2 let b = a + 2;
3 console.log(b);
```

このプログラムを実行すると，1 行目の値 1 は代入の右辺で現れ，左辺の変数 a に格納される．その後，2 行目の代入の左辺の式中で変数 a から取り出される．このような通過場所を記憶する追跡子を値 1 に付与すると図 2.1 のようになる．同様に，2 行目の式 a + 2 の評価結果として生成された値 3 は，2 行目の左辺の変数 b に格納され，その後，3 行目で変数 b から取り出され，関数 console.log の実引数となる．このような通過場所を記憶する追跡子を値 3 に付与すると図 2.2 のようになる．また，各追跡子にはそれが付与された時刻が暗黙的に含まれるため，場所を通過した順番の情報も含まれる．

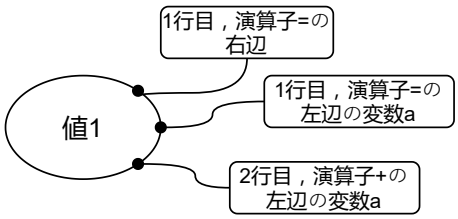

図 2.1　通過場所を記憶する追跡子が付与された値 1

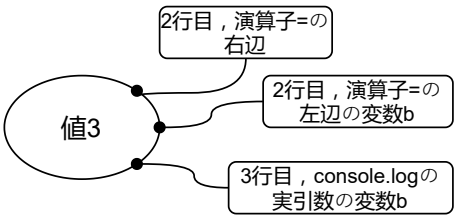

図 2.2　通過場所を記憶する追跡子が付与された値 3

値の生成源の値（生成のために使用された値）を記憶する追跡子

本手法における，値 v_{out} の生成源の値 v_{in} の定義を以下に示す

定義 値 v_{in} を使用した演算の結果の値として，値 v_{out} が新たに生成される場合（**定義 1**），または，値 v_{in} は条件文（if，for，while）の条件式の評価結果の値であり，その条件文の分岐のブロック内の式の評価結果として値 v_{out} が新たに生成される場合（**定義 2**），値 v_{in} は値 v_{out} の生成源の値である．

定義 1 について，例えば式 1 + 2 の評価結果の値 3 は，値 1 と値 2 を使用した演算 + の結果として新たに生成された値であるので，値 1，2 は値 3 の生成源の値である．このとき値 3 に生成源の値を記憶する追跡子を付与すると図 2.3 のようになる．同様に，配列参照式 arr[5] を評価した結果，配列外参照によって値 undefined が得られた場合，この値 undefined は配列参照演算のオペランドの式 arr の値（配列の値）と添字の値 5 から新たに生成された値であるので，配列 arr の値と値 5 は値 undefined の生成源の値である．このとき値 undefined に生成源の値を記憶する追跡子を付与すると図 2.4 のようになる．一方で，式 arr[5] を評価した結果，すでに格納されていた値（仮に 42 とする）が取り出された場合には，配列 arr の値と値 5 は値 42 の生成源の値ではない．

定義 2 について，ソースコード 2.2 について考えると，2 行目の条件式 n % 2 == 0 の評価結果が値 true だった場合，3 行目が実行されて値（文字列）である'even' が新たに

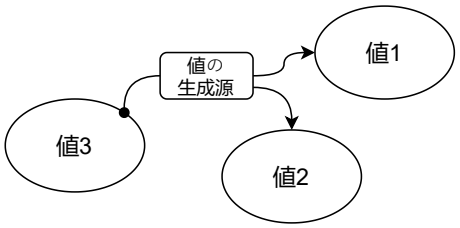

図 2.3　値の生成源の値を記憶する追跡子が付与された値 3

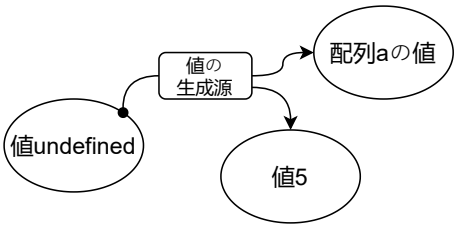

図 2.4　値の生成源の値を記憶する追跡子が付与された値 undefined

生成される（本研究ではこのようなリテラル文字列はその場において生成されるものとした）．よって，値 true は値'even' の生成源である．このとき値'even' に生成源の値を記憶する追跡子を付与すると図 2.5 のようになる．同様に，条件式 n % 2 == 0 の評価結果が値 false だった場合，6 行目が実行されて値（文字列）'odd' が新たに生成されるので，値 false は値'odd' の生成源である．このとき値'odd' に生成源の値を記憶する追跡子を付与すると図 2.6 のようになる．

ソースコード 2.2　値の生成源の値と条件文（その 1）

```
1 function f(n) {
2   if(n % 2 == 0) {
3     return 'even';
4   }
5   else {
6     return 'odd';
7   }
8 }
```

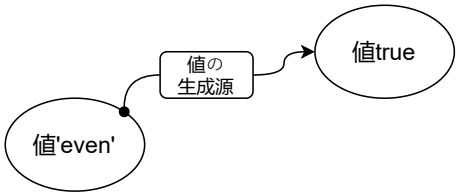

図 2.5　値の生成源の値を記憶する追跡子が付与された値'even'

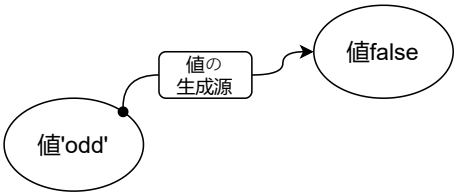

図 2.6　値の生成源の値を記憶する追跡子が付与された値'odd'

一方で，ソースコード 2.3 について考えると，3 行目の条件式 n % 2 == 0 の評価結果が値 true だった場合，値 true はその if 文のブロック内 4 行目で新たに生成される値'even' の生成源となるが，条件式 n % 2 == 0 の評価結果が値 false だった場合，実行される 7 行目では，2 行目で既に生成されていた値'odd' を変数 str_odd から取り出すのみのため，値 false は値'odd' の生成源ではない．

ソースコード 2.3　値の生成源の値と条件文（その 2）

```
1 function f(n) {
2   const str_odd = 'odd';
3   if(n % 2 == 0) {
4     return 'even';
5   }
6   else {
7     return str_odd;
8   }
9 }
```

ユーザ指定の文字列を記憶する追跡子

ユーザ指定の文字列については，例えばソースコード 2.1 で変数 a にアクセスしている場所を通過した値に文字列 *refa* を記憶する追跡子を付与するようユーザが指示した場合，実行時にこの場所を通過する個々の値 1 にその追跡子が付与される（図 2.7）．また，追跡子文字列は 1 つの付与ポイントで複数個付与可能とする．

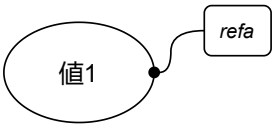

図 2.7　ユーザ指定の文字列を記憶する追跡子が付与された値 1

文字列を記憶する追跡子を付与することの指示は，プログラムの字面上から行うことにした．例えば，任意の式 *exp* の評価結果の値に対して追跡子文字列 *marker* を付与するための特殊形式 MARK(*exp*, *marker*) を導入すると，ソースコード 2.4 の 2 行目のように記述することができる．

ソースコード 2.4　文字列を記憶する追跡子の付与を指示したプログラム（その 1）

```
1 let a = 1;
2 let b = MARK(a, 'refa') + 2; // 変数a を通過した個々の値に対して追跡子文字列'refa'を
      付与
3 console.log(b);
```

プログラムの字面上から追跡子文字列の付与を指示するにあたって，追跡子文字列を付与する対象の値には自由度がある．特殊形式 MARK(*exp*, *marker*) について，追跡子文字列 *marker* を付与する対象の値は式 *exp* の評価結果の値（バリエーション 1，静的な追跡子付与）と前述したが，このほかにも以下のようなバリエーションがある．

- 式 *exp* 全体の評価結果の値のほかに，その部分式の評価結果の値にも追跡子文字列 *marker* を付与する（バリエーション 2，静的な追跡子付与）
- 式 *exp* 全体の評価結果の値のほかに，その式を評価している間に評価された他のすべての式の評価結果の値にも追跡子文字列 *marker* を付与する（バリエーション 3，動的な追跡子付与）

例えばソースコード 2.5 では，5 行目の関数呼出式 `twice(a)` を第一引数として `MARK` を使用している．このとき，追跡子文字列`'result'` が付与される対象の値は，バリエーション 1 の場合，式 `twice(a)` の評価結果の値 6 のみである．バリエーション 2 の場合，式 `twice(a)` 全体の評価結果の値 6 のほかに，その部分式 `twice`，`a` の評価結果の値（関数 `twice` の値，値 3）にも追跡子文字列`'result'` が付与される．バリエーション 3 の場合，式 `twice(a)` 全体の評価結果の値 6 のほかに，式 `twice(a)` の部分式 `twice`，`a` の評価結果の値と，`twice(a)` の関数呼出によって評価される，`twice` 関数内 2 行目の式 `x * 2` 中の値 2 にも追跡子文字列`'result'` が付与される．

ソースコード 2.5　文字列を記憶する追跡子の付与を指示したプログラム（その 2）

```
1 const twice = (x) => {
2   return x * 2;
3 }
4 let a = 3;
5 let b = MARK(twice(a), 'result');
```

本手法では，このようなバリエーションをそれぞれ別々の特殊形式として区別して使用できるということにした．

また，1 つの値に同一の追跡子文字列が付与される場合でも，付与された時刻が異なる場合は，個々の追跡子として付与する．そのため，例えばソースコード 2.5 をバリエーション 3 の追跡子文字列付与方式で実行した際，4 行目で発生する値 3 には，5 行目の関数呼出式 `twice(a)` の実引数の変数 `a` から取り出された時刻，1 行目の仮引数 `x` に渡された時刻，2 行目の式 `x * 2` 中の変数 `x` から取り出された時刻で追跡子文字列`'result'` が付与される．

2.1.2　値と追跡子によるグラフ構造の形成

値と付与された追跡子はグラフ構造を形成する．例えば，ソースコード 2.1 を実行後に得られる，実行時の全ての値及び付与された追跡子は図 2.8 のようなグラフ構造を形成する．

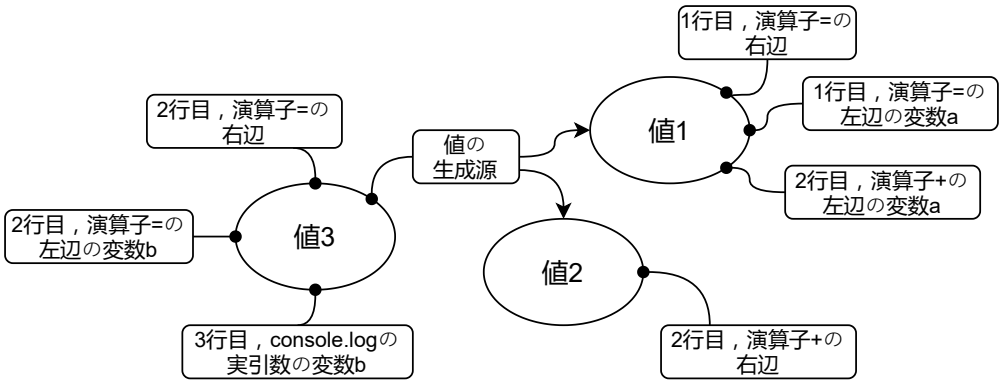

図 2.8　値とその追跡子のグラフ構造

2.2　クエリ言語

値への追跡子付与によって得られたグラフ構造から，有用な情報を取り出すためのクエリ言語について説明する．

2.2.1　設計方針

クエリ言語を設計するにあたっては，必要最低限の基本クエリ機能のみを導入すること，かつ，基本クエリ機能の組み合わせによって，より高度で複雑なクエリを実現できることに注意した．

まず，値が持つ追跡子を確認するクエリが考えられ，これを基本クエリとして採用する．本手法で導入した追跡子は 3 種類のため，このクエリは確認する追跡子のそれぞれについて 3 種類が考えられる（クエリ 1，2，3）．

クエリによる値のフィルタリングの方法として，追跡子に関する条件を使用するものと，そうでないものがある．前者に関して，追跡子に関する最も基本的な条件とは，追跡子を「持つか持たないか」であると考える．本クエリ言語では，この考えに基づいて，ある追跡子を指定し，その追跡子を持つ/持たない値のみを取り出すためのクエリを採用する．このクエリに関しても，条件で使用する追跡子のそれぞれについて 3 種類が考えられる（クエリ 4，5，6）．後者に関しては，一般的なフィルタリング機能として採用する．このクエリは，デバッグ対象のプログラムと同じ言語で記述された述語を引数として，その述語を満たす値のみを返すようにする．

クエリ 1〜6 の意味や有用性について，続く 2.2.2 節で議論する．

2.2.2 クエリ機能一覧

クエリ機能 1：値を指定し，その値が通過したプログラム中の場所を記憶する追跡子を取り出すクエリ

ある値が通過したプログラム中の場所を記憶する追跡子を取り出すと，その値が通過したプログラム中の場所とその順序（値の流れ）が分かる．この情報は，例えばプログラムの字面上の場所間に線を引くなどといった視覚的な情報として，値の流れの理解を支援するために有用であると考えられる．

クエリ機能 2：値を指定し，その値の生成源の値を記憶する追跡子を取り出すクエリ

値の生成源の値を記憶する追跡子を取り出すと，その値が他のどの値から生成されたのかが分かる．このクエリは，意図しない値の生成源を遡り，その値が生成された原因（バグの原因）を特定する際に有用であると考えられる．

クエリ機能 3：値を指定し，その値が持つユーザ指定の文字列を記憶する追跡子を取り出すクエリ

このクエリによって取り出された，値に付与された追跡子文字列の意味は，ユーザが追跡子文字列を付与するよう指定した意図に依存する．そのため，このクエリによって，その値がユーザの関心事を満たす値かどうかを確認することができる．

クエリ機能 4：プログラム中の場所を通過したことを記憶する追跡子を指定し，その追跡子を持つ/持たない値群を取り出すクエリ

このクエリによって，ある場所を通過した/しなかった値群を取り出すことができる．これは，意図しない値が生成された原因を特定するデバッグの起点となる値を取得するために有用であると考えられる．

クエリ機能 5：生成源の値を記憶する追跡子を指定し，その追跡子を持つ/持たない値群を取り出すクエリ

このクエリによって，指定した値から生成された/されなかった値群を取り出すことができる．これは，クエリ機能 2 の逆の操作であるといえる．例えば，すでに分かっている，バグを引き起こすかもしれない怪しい値を指定して，その値の影響を受ける値群を調べる場合などに有用であると考えられる．

クエリ機能 6：ユーザ指定の追跡子文字列を指定し，その追跡子を持つ/持たない値群を取り出すクエリ

このクエリでは，ユーザ指定の追跡子文字列を持つ/持たない値群を取り出すことができる．このクエリによって取り出された値の意味は，ユーザが追跡子文字列を付与するよう指定した意図に依存する．そのため，このクエリはユーザの様々な関心事に応じて値を取り出すことができる点で強力なクエリであると考えられる．

2.2.3 その他のクエリ機能

クエリ機能 7：ユーザ指定の追跡子文字列に関する条件を指定し，その条件を満たす値群を取り出すクエリ

クエリ機能 7 では，同付与時刻に付与された追跡子文字列に対する条件を使用したクエリ（$mkrEveryTime$，$mkrSomeTime$）を記述できる．これにより，クエリ機能 6 では表現できなかった，ユーザ指定追跡子文字列に関するクエリを記述することができる．ソースコード 2.6 実行時に 4 行目で発生する値 0 が追跡子付与ポイントを通過する時刻と，その時刻において付与されるユーザ指定追跡子文字列は表 2.1 のようになる．

ソースコード 2.6　ユーザ指定追跡子文字列付与を使用したプログラム

```
1 function f(x) {
2   return MARK(x, 'in_f'); // バリエーション 1方式での追跡子文字列付与を指示
3 }
4 let a = 0;
5 MARK_DY(f(a), 'call_f'); // バリエーション 3方式での追跡子文字列付与を指示
6 MARK(a, 'out_f'); // バリエーション 1方式での追跡子文字列付与を指示
7 a = MARK(1, 'one'); // バリエーション 1方式での追跡子文字列付与を指示
```

クエリ機能 6 によって値 0 を取得する際には，以下のようなクエリを実行できる．

- 追跡子文字列'in_f' を持つ値を取得
- 追跡子文字列'call_f' を持つ値を取得
- 追跡子文字列'out_f' を持つ値を取得
- 追跡子文字列'one' を持たない値を取得

クエリ機能 7 では，以下のように，同付与時刻に付与された追跡子文字列の条件から値 0 を取得することができる．

- $mkrEveryTime$('in_f' ⇒ 'call_f')

表 2.1　値 0 の追跡子付与ポイント通過時刻と付与される追跡子文字列

付与時刻	通過場所	付与される追跡子文字列
1	4 行目右辺の式 0	[]
2	4 行目左辺の変数 a	[]
3	5 行目関数呼出の実引数 a	['call_f']
4	1 行目仮引数 x	['call_f']
5	2 行目変数 x	['call_f', 'in_f']
6	5 行目関数呼出 f(a)	['call_f']
7	6 行目変数 a	['out_f']

  - すべての付与時刻で，'in_f' が付与されているなら'call_f' も付与されている値を取得する．表 2.1 について，付与時刻 1, 2, 3, 4, 7 では，そもそも'in_f' が付与されていないため，条件を満たす．また，付与時刻 5 では，'in_f' が付与されているが，'call_f' も付与されているため，条件を満たす．よって，すべての付与時刻で条件を満たす．

- $mkrSomeTime$('in_f' ∧ 'call_f')
  - 'in_f' と'call_f' が同時に付与されている付与時刻がある値を取得する．表 2.1 について，付与時刻 1, 2, 3, 4, 7 では，'in_f' と'call_f' は同時に付与されていないため，条件を満たさない．しかし，付与時刻 5 では，'in_f' と'call_f' は同時に付与されており，条件を満たす．よって，条件を満たす付与時刻が存在する．
- $mkrEveryTime$(¬'call_f' ∨ ¬'out_f') または
  $mkrEveryTime$(¬('call_f' ∧ 'out_f'))
  - すべての付与時刻で，'call_f' が付与されていないか'out_f' が付与されていない値を取得（'call_f' と'out_f' が同付与時刻に付与されることがない値を取得）する．表 2.1 について，付与時刻 1, 2 では，'call_f' も'out_f' も付与されていないため，条件を満たす．また，付与時刻 3〜6 では，'call_f' が付与されているが'out_f' は付与されていないため，条件を満たす．付与時刻 7 では，'out_f' が付与されているが'call_f' は付与されていないため，条件を満たす．よって，すべての付与時刻で条件を満たす．

クエリ機能 8：集合演算クエリ機能

値の集合に対する集合演算（和集合，積集合の計算）を行うクエリを導入する．この演算は，値の集合の部分集合を返すクエリであるクエリ 2，4，5，6，7 の結果に対しても施すことができる．これにより，クエリの組み合わせによるより高度で複雑なクエリを実現できる．

第 3 章

提案手法を活用したツールの実装

この章では，本研究が提案する手法を活用したデバッグツールの実装について説明する．このツールはブラウザアプリとして実装されており，JavaScript プログラムのデバッグを行うことができる．

3.1 完成したツールの全体像と構成

完成したツールの全体像を図 3.1 に示す．このツールは，7 つの要素から構成されている．

要素 A は，解析対象の JavaScript プログラムを記述するためのエディタである．要素 B のボタンは，要素 A のエディタに記述したプログラムを実行するためのボタンであり，その実行結果の標準出力は，要素 C のターミナルに表示される．また，要素 D のボタンから，要素 C のターミナルの出力内容を消去できる．

要素 E，F，G は，要素 A のエディタに記述したプログラムをデバッグするためのものである．要素 E は，クエリによるデバッグのためのコンソールである．このコンソールは，デバッグ中以外は無効になっている．要素 F のボタンを押すと，要素 A のエディタに記述したプログラムのデバッグを開始し，要素 E のコンソールが有効になる．プログラムのデバッグ実行中，要素 E にはプロンプトが表示され，クエリを記述することができる．クエリの実行結果は，要素 E のターミナル上に文字列情報として提供される場合と，要素 A のエディタ上にグラフィカル情報として提供される場合（図 3.2）がある（詳細は後述）．要素 A のエディタ上に提供するグラフィカル情報は，デバッグ対象プログラムの静的解析時に取得した各部分プログラムの場所情報を使用している．そのため，デバッグ中にエディタ上のプログラムが変更されることで場所情報の整合性が無くなってしまわないように，デバッグ実行中はエディタは読み取り専用モードになっている．最後に，要素 G のボタンから，デバッグモードを終了できる．

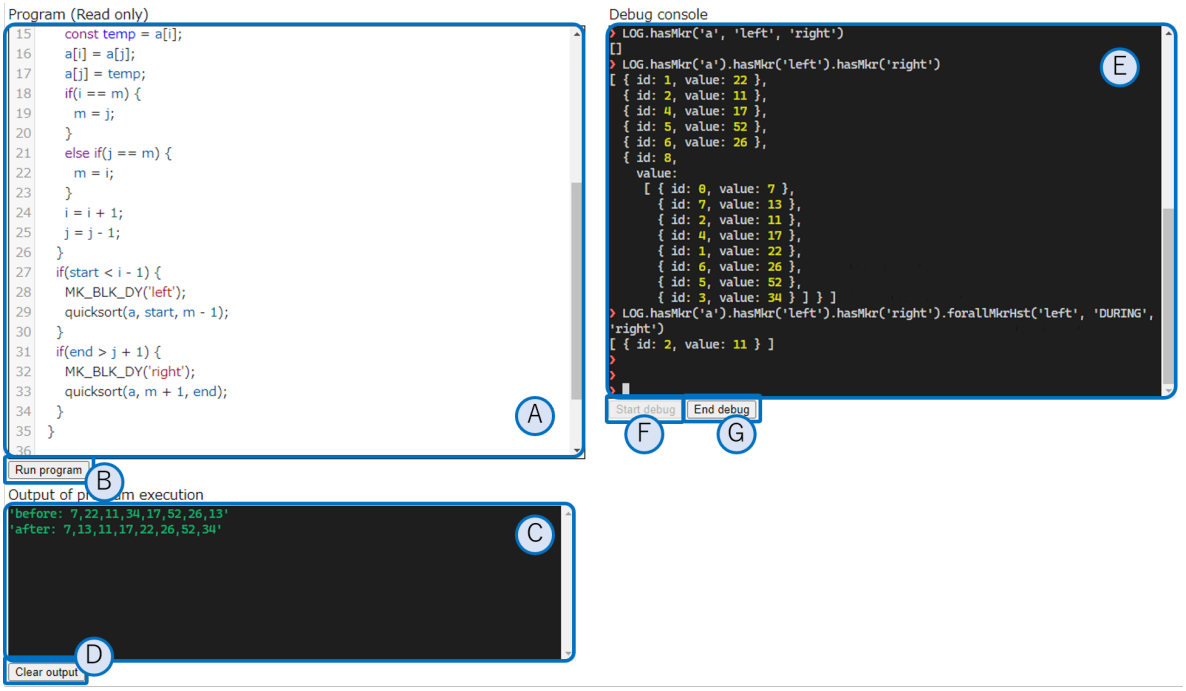

図 3.1　完成したツールの全体像（デバッグ中の様子）

```
1  function twice(arr) {
2    let ans = [];
3    for (let i = 0; i <= arr.length; i = i + 1) { // bug
4      ans[i] = arr[i] * 2;
5    }
6    return ans;
7  }
8  function sum(arr) {
9    let ans = 0;
10   for (let i = 0; i < arr.length; i = i + 1) {
11     ans = ans + arr[i];
12   }
13   return ans;
14 }
15 let arr = [2, 7, 1, 8, 2, 8];
16 let arr_twice = twice(arr);
17 let arr_sum = sum(arr_twice);
18 console.log(arr_sum);
```

図 3.2　グラフィカル情報の提供の例

3.2 値への追跡子付与の実現

この節では，本ツールを実現するにあたって，提案手法の 1 つである値への追跡子の付与をどのように実装したかについて説明する．

3.2.1 設計方針

追跡子情報の取得は，プログラム変換によって実現している（図 3.3）．このため，本手法は変換器が実装できれば任意の言語に対して適用でき，変換後のプログラムは既存の言語処理系で実行可能である．デバッグ対象のプログラムは，変換器によって追跡子付与能力付きのプログラムに変換される．この変換後のプログラムを既存の実行環境によって実行することにより，元のデバッグ対象プログラムの計算と追跡子による値の追跡が行われ，結果として追跡子付きの値群が得られる．

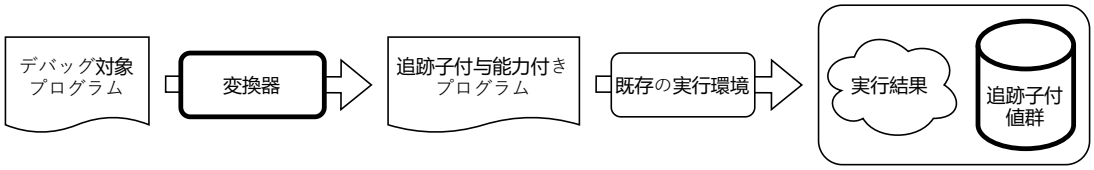

図 3.3 デバッグ対象プログラムの追跡子情報取得メカニズムの全体像

本手法では，プログラム実行時のすべての値は追跡子を自身の保持する能力を持つ．これを実現するために，実装では，追跡子保持能力付きの値を表すデータ構造を導入し，プログラム実行時のすべての値はこのデータ構造のインスタンスとなるようにする．また，実行時に発生したすべての追跡子付きの値はプログラムの実行結果とともに出力する必要があるため，実行時に発生したすべての追跡子付きの値を格納しておくための大域コンテナ *values* を変換後のプログラムに導入し，これを実行結果とともに出力するデータとする．

本手法では，追跡子の付与は，プログラム中の式が評価され値が得られるタイミングで行う．つまり，「各式について，その式を評価して得られた値をそのまま返す前に，その値に追跡子を付与してから返す」ようにしなければならない．これを実現するためには，値を受け取りその値に追跡子を付与して返すような関数を用意し，デバッグ対象のプログラム中のすべての式を，その関数の引数として与えればよい．よって，基本的には，任意の式 *exp* に対して，追跡子付与のための関数 *mark* を使用して以下のような変換を行う．

$$exp \xrightarrow{\text{変換}} mark(exp, \langle \text{付与する追跡子に与える情報} \rangle)$$

ここで，関数 *mark* は，第 1 引数に与えられた式の評価結果の値に対して，第 2 引数以降が

意味する追跡子を付与し, その値を返す．これにより, 例えば式 `1 + 2` は, `MARK(MARK(1, ...) + MARK(2, ...), ...)` などと変換される．ここで，演算 + は，その言語における値同士の加算を行うものであり，`MARK` 関数の返値である追跡子保持能力付きの値に対しては互換性がなく，変換後のプログラムを実行するとエラーとなる．そこで，受け取った値に追跡子を付与して追跡子保持能力付きの値を返すのではなく，元の言語における値を返す関数 $mark'$ を導入し，その言語におけるプリミティブな演算を行うための値となる式に対する変換の際には，$mark'$ を使用するようにする．これにより，式 `1 + 2` は，`MARK(MARK_D(1, ...) + MARK_D(2, ...), ...)` などと変換され，このプログラムは正しく実行することができる．

関数 $mark$，$mark'$ の第 1 引数となる値には 2 つの可能性が考えられる．1 つ目の可能性は，追跡子保持能力付きの値を表すデータ構造のインスタンスの値である．この場合，それぞれの関数では，その値に対して追跡子を付与し，その値，または元の言語における値を返す．もう 1 つの可能性は，元の言語における値である．プログラム実行時のすべての値は追跡子を自身に保持する能力を持つ，つまり追跡子保持能力付きの値を表すデータ構造のインスタンスの値であるはずなので，ここでの元の言語における値は，演算によって新たに生成された値である．この場合，それぞれの関数では，その値を追跡子保持能力付きの値とし，その値に生成源の値を記憶する追跡子を付与して返す．生成源の値の情報は，第 2 引数以降の付与する追跡子に与える情報を使用する．

プログラム変換において，関数 $mark$，$mark'$ の第 2 引数以降の，付与する追跡子に与える情報は，以下の通りである．

1. 式 *exp* のプログラム中の場所の情報．つまり，式 *exp* の文字列の開始位置と終了位置（行数，列数）
2. 式 *exp* の評価結果の値の生成源となる可能性のある値．つまり式 *exp* の部分式の評価結果の値
3. 式 *exp* に対して付与するよう指定されたユーザ指定の文字列

1. について，これはデバッグ対象のプログラムの字面上の情報から取得可能である．

2. について，静的解析であるプログラム変換の段階において，プログラム実行時に動的に決まる式の評価結果の値を取得することはできない．しかし，式の評価結果の値の生成源となる可能性のある値を生み出す式は，静的に定めることができる．例えば，式 `a[i]` において，この式を評価することによって得られる値はこの式を実行してみないと分からないが，式の字面から，式 `a[i]` の評価結果の値は，式 `a` の評価結果の値と，式 `i` の評価結果の値から新たに生成される可能性があることが分かる．そこで，この例の場合，式 `a`，`i` への追跡子付与の処理のついでに，それらの評価結果の値を適当な ID と紐づけて大域的な表 *ref* に記録しておき，式 `a[i]` への追跡子付与の処理の際に，それらの ID を使用し

て表 *ref* から値を必要に応じて取り出すようにすればよい．

3. について，まず，ユーザ指定の追跡子付与のための特殊形式は，関数呼出式として，既存の JavaScript の言語の枠組みの中で実現する．追跡子付与対象の範囲が静的に定まるバリエーション 1，2 の場合は，プログラム字面上の，特殊形式を表現した関数呼出式の情報から静的に取得可能である．バリエーション 3 の場合，プログラム実行時に動的に決まる評価される式を静的に決定することはできない．しかし，式の評価開始と終了の位置に対しては字面で定めることができる．よって，変換後のプログラムに大域的なスタック *dynamicMarkers* を用意し，式の評価開始位置で，付与すべきユーザ指定追跡子を *dynamicMarkers* に push する．そして，式を評価している間，追跡子を付与する処理の際は，*dynamicMarkers* に格納されている追跡子も付与対象に含める．そして，式の評価後に，*dynamicMarkers* の内容を式評価直前の状態にリストアし，最後に式の評価結果の値を返せばよい．よって，式 *exp* を以下のように変換する．ここで，*save*() は *dynamicMarkers* の状態を，式評価後復帰用の大域スタック *stateStackExp*（変換後のプログラムに導入）にセーブしておくための関数，*push*(*marker*) は *dynamicMarkers* に *marker* を push するための関数，*restore*() は *stateStackExp* から *dynamicMarkers* の状態をリストアするための関数である．

$$exp \xrightarrow{\text{変換}} (save(),\ push(marker),\ tmp = mark(exp, \ldots),\ restore(),\ tmp)$$

複数の操作を 1 つの式として表現するために，JavaScript のカンマ演算子を使用している．また，式全体の評価結果の値は *exp* の評価結果の値になることが必要，かつ，式 *exp* は追跡子格納以前の状態に *dynamicMarkers* をリストアする前に行う必要がある．これを実現するために，ユーザ指定追跡子付与範囲中に評価した式 *exp* の値は一時変数 *tmp* に格納しておき，カンマ演算子の最後の式として変数 *tmp* から取り出す事によって，式全体の結果の値としている．

しかし，この変換では，変換後の式全体が正常に評価されることを前提としており，式 *exp* の評価中に例外が発生した場合を考慮していない．特に，式 *exp* 評価中に発生した例外が外部でキャッチされると，*dynamicMarkers* をリストアせずに実行が進んでしまい，結果として意図しない値に不適切な追跡子が付与されてしまう．この場合の正しい挙動は，*dynamicMarkers* と *stateStackExp* の状態が，例外処理の try ブロックにプログラム実行が進んだ時点での状態へとリストアされることである．そこで，例外処理前に，*dynamicMarkers* の状態と，*stateStackExp* の状態を push し，例外キャッチ時に，*dynamicMarkers*，*stateStackExp* を例外処理前の状態にリストアするための例外発生時の復帰用大域スタック *stateStackExcep* を変換後のプログラムに導入する．

したがって，*mark* 関数の仕様は以下の通りである．

入力 – 追跡子付与対象の値（元の言語の値，または，追跡子保持能力付きの値）

– 付与対象の追跡子（場所，生成源になりうる値，ユーザ指定の追跡子文字列）
– 追跡子付与対象の値を大域的な表に記録するための ID

出力 追跡子保持能力付きの値

処理 1. 追跡子付与対象の値が元の言語の値の場合，その値から新たに追跡子保持能力付きの値 v_{new} を作成し，v_{new} に生成源になりうる値を記憶する追跡子を付与する．最後に，すべての追跡子付きの値を格納しておく大域コンテナ $values$ に v_{new} を格納する．また，以降の手順では，v_{new} を追跡子付与対象の値とする．
2. 追跡子付与対象の値に対して，通過場所を記憶する追跡子，ユーザ指定の追跡子文字列，$dynamicMarkers$ 内に格納されている追跡子をそれぞれ付与する．
3. 追跡子付与対象の値を ref に入力の ID で記録する．

また，$mark'$ 関数は，$mark$ 関数の返値の追跡子保持能力付きの値から，元の値を取り出して返せばよい．

3.2.2 変換器の実装

変換器は，Javascript プログラムのトランスパイラ Babel [1] のプラグインとして実装した．行数は約 700 行である．

変換器の内部構造を図 3.4 に示す．変換器は，入力のデバッグ対象プログラムを受け取り，そのプログラムの意味に相当する抽象構文木（AST）を生成する．そして，AST を走査し，変形する．最後に，変形後の AST からプログラムを生成する．プログラムからの AST の生成と，AST からのプログラムの生成は，Babel のライブラリで用意されている API をそのまま使用して実現した．また，AST 変換プログラムは，Babel のライブラリ API を使用して実装を行った．

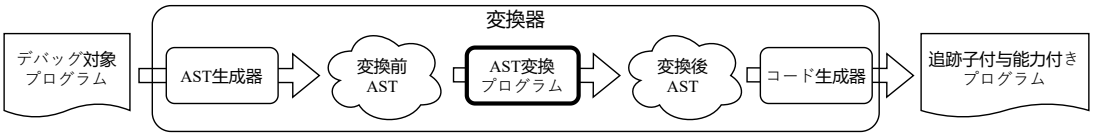

図 3.4 変換器の内部構造

設計における $mark$, $mark'$ 関数は，それぞれ_PASS，_GETVAL 関数として実装した．また，追跡子付与対象の値を一時的に格納する大域的な表 ref は配列_REF として実装した．また，式を通過した値に対してユーザ指定の追跡子を付与することを指示する特殊形式は，バリエーション 1，2，3 を，それぞれ関数 MK_EXP(*exp*，*marker*)，MK_EXP_ST(*exp*，*marker*)，MK_EXP_DY(*exp*，*marker*) として実装した．さらに，ブロックに対して，その

ブロック中に含まれる式を通過した値にユーザ指定の追跡子を付与することを指示する特殊形式を関数 MK_BLK_ST(*marker*) として実装し，そのブロック内を実行中に評価された値にユーザ指定の追跡子を付与することを指示する特殊形式を関数 MK_BLK_DY(*marker*) として実装した．

実装における_PASS，_GETVAL 関数のシグネチャは以下の通りである．

引数
- 追跡子付与対象の値（JavaScript の値，または，オブジェクトとして表現された追跡子保持能力付きの値）
- 場所を表す数値（式開始位置の行・列数，終了位置の行・列数）
- 生成源になりうる値の配列
- ユーザ指定の追跡子文字列（文字列）を格納する配列
- 追跡子付与対象の値を_REF に格納する際の ID（文字列）

返値
- オブジェクトとして表現された追跡子保持能力付きの値（_PASS 関数の場合）
- JavaScript の値（_GETVAL 関数の場合）

実際にこれらの関数を利用して，ソースコード 3.1 のプログラムは，ソースコード 3.2 のプログラムに変換される．

ソースコード 3.1　プログラム変換例（変換前）

```
1 a + MK_EXP(1, 'hoge')
```

ソースコード 3.2　プログラム変換例（変換後）

```
1  _PASS(
2    _GETVAL( // 式a の評価結果の値は JavaScript の値同士の演算子+で使用されるので,関数
         _GETVAL を使用
3      a,
4      1, 0, 1, 1, // 式a の場所を表す数値（1 行 0 列目から 1 行 1 列目）
5      [], // 式a の評価結果の値の生成源となりうる値
6      [], // ユーザ指定の追跡子文字列を格納する配列
7      'id1' // 式
           a の評価結果の値を_REF に格納する際の ID．変換器による静的解析の際に自動で割り振られる

8    ) +
9    _GETVAL( // 式 1の評価結果の値はJavaScript の値同士の演算子+で使用されるので,関数
         _GETVAL を使用
10     1,
11     1, 4, 1, 5, // 式 1の場所を表す数値(1行 4列目から 1行 5列目)
12     [], // 式 1の評価結果の値の生成源となりうる値
13     ['hoge'], // ユーザ指定の追跡子文字列を格納する配列
```

```
14      'id2' // 式 1の評価結果の値を
            _REF に格納する際の ID．変換器による静的解析の際に自動で割り振られる
15    ),
16    1, 0, 1, 5, // 式a + 1の場所を表す数値(1行 0列目から 1行 5列目)
17    [_REF['id1'], _REF['id2']], // 式a + 1の評価結果の値の生成源となりうる値
18    [], // ユーザ指定の追跡子文字列を格納する配列
19    'id3' // 式a + 1の評価結果の値を
          _REF に格納する際の ID．変換器による静的解析の際に自動で割り振られる
20  )
```

本変換器は，JavaScript の基本的な構文への対応を行った．本変換器が対応している構文と，その変換方法について以下に示す．ただし，以下の変換例では，場所を表す数値は loc と表し，ユーザ指定の追跡子文字列を格納する配列，及び追跡子付与対象の値を_REF に格納する際の ID は省略する．

数値，文字列，真偽値，null，undefined，NaN，this

これらのアトム・シンボルは，その式の外の文脈に応じて関数_PASS または_GETVAL に渡して変換する（変換例では_PASS 関数）．

ソースコード 3.3　アトム・シンボル（変換前）

```
1  1
2  'hoge'
3  true
4  null
5  undefined
6  NaN
```

ソースコード 3.4　アトム・シンボル（変換後）

```
1  _PASS(1, loc, [])
2  _PASS('hoge', loc, [])
3  _PASS(true, loc, [])
4  _PASS(null, loc, [])
5  _PASS(undefined, loc, [])
6  _PASS(NaN, loc, [])
```

配列式，オブジェクト式

配列・オブジェクト式では，まずそれらの式が持つ要素・プロパティの式を再帰的に変換する．そして，配列・オブジェクト式全体を，その式の外の文脈に応じて関数_PASS または_GETVAL に渡して変換する（変換例では_PASS 関数）．

配列式，オブジェクト式の値は，その値が持つ要素・プロパティから生成されうる．そのため，生成源となりうる値の配列に，それらの値を含める．ここで，関数 $id(\_)$ は，式 exp から，その評価結果の値を_REF に格納する際の ID へのマッピングである．また，ハット付きの式 $\widehat{exp}$ は，式 exp の部分式が適切に変換され，かつ式 exp が_PASS 関数によって変換済みであることを表す．

ソースコード 3.5　配列・オブジェクト式（変換前）

```
1 [exp_1, ..., exp_n]
2 {prop_1 : exp_1, ..., prop_n : exp_n}
```

ソースコード 3.6　配列・オブジェクト式（変換後）

```
1 _PASS([exp_1^, ..., exp_n^], loc, [_REF['id(exp_1)'], ..., _REF['id(exp_n)']])
2 _PASS({prop_1 : exp_1^, ..., prop_n : exp_n^}, loc, [_REF['id(exp_1)'], ..., _REF['id(exp_n)'
      ]])
```

変数式

変数式は，その式の外の文脈に応じて関数_PASS または_GETVAL に渡して変換する（変換例では_PASS 関数）.

ソースコード 3.7　変数式（変換前）

```
1 a
```

ソースコード 3.8　変数式（変換後）

```
1 _PASS(a, loc, [])
```

2 項演算式

2 項演算式では，まずオペランドの式を再帰的に変換する．そして，式全体を，その式の外の文脈に応じて関数_PASS または_GETVAL に渡して変換する（変換例では_PASS 関数）. また，2 項演算のオペランドは JavaScript の値でなければならないので，式 exp_1, exp_2 は，_GETVAL 関数によって変換する．ここで，チェック付きの式 $\widetilde{exp}$ は，式 exp の部分式が適切に変換され，かつ式 exp が_GETVAL 関数によって変換済みであることを表す.

2 項演算式の値は，その式の演算のオペランドの値から生成されうる．そのため，生成源となりうる値の配列に，それらの値を含める.

ソースコード 3.9　2 項演算式（変換前）

```
1 exp_1 + exp_2
```

ソースコード 3.10　2 項演算式（変換後）

```
1 _PASS(exp_1~ + exp_2~, loc, [_REF['id(exp_1)'], _REF['id(exp_2)']])
```

配列参照式

配列参照式では，まずオペランドの式（参照先配列の式と参照場所の式）を再帰的に変換する．その後，配列参照式全体を，外の文脈に応じて関数_PASS または_GETVAL に渡

して変換する（変換例では_PASS 関数）．また，配列参照のオペランドは JavaScript の値でなければならないので，式 exp_1, exp_2 は_GETVAL 関数によって変換する．

配列参照式の値は，その式の演算のオペランドの値から生成されうる．そのため，生成源となりうる値の配列に，それらの値を含める．

ソースコード 3.11　配列参照式（変換前）

```
1 exp1[exp2]
```

ソースコード 3.12　配列参照式（変換後）

```
1 _PASS(~exp1~[~exp2~], loc, [_REF['id(exp1)'], _REF['id(exp2)']])
```

メンバ参照式

メンバ参照式では，まず呼出オブジェクトの式 *exp* を再帰的に変換する．メンバ参照は，式 *exp* の評価結果の元のオブジェクトの値に対して行わなければいけないので，式 *exp* は_GETVAL 関数によって変換する．また，メンバ名 *prop* は静的な識別子のため，変換しない．最後に，メンバ参照式全体を，外の文脈に応じて関数_PASS または_GETVAL に渡して変換する（変換例では_PASS 関数）．

メンバ参照式の値は，その呼出元のオブジェクトの値から生成されうる．そのため，生成源となりうる値の配列に，それらの値を含める．

ソースコード 3.13　メンバ参照式（変換前）

```
1 exp.prop
```

ソースコード 3.14　メンバ参照式（変換後）

```
1 _PASS(~exp~.prop, loc, [_REF['id(exp)']])
```

関数呼出式

関数呼出式では，まずオペレータとオペランドの式を再帰的に変換する．JavaScript の関数値に対して関数呼出を行わなければならないため，オペレータは_GETVAL 関数によって変換する．また，オペランドの変換のために使用する関数は，オペレータの関数がユーザ定義の関数（変換対象に含まれる関数）である場合には_PASS，オペレータの関数が組み込み関数，または外部の関数（API 関数など）の場合（変換対象外の関数）である場合には_GETVAL を使用すべきである．しかし，この判断を静的解析時にすべて正しく行うことは不可能である．そこで，変換器としては，オペランドの式は決め打ちで_PASS 関数

によって変換することとした．そして，対象の式を_GETVAL 関数で変換することを指示するための特殊形式を関数 DETACH_MKR(*exp*) として用意した．これにより，組み込み関数や外部の関数の実引数として渡す式に対して，ユーザの判断で適切な変換を指示できるようにした．

最後に，関数呼出式全体を，外の文脈に応じて関数_PASS または_GETVAL に渡して変換する（変換例では_PASS 関数）．

関数呼出式の値は，オペレータとオペランドの値から生成されうる．そのため，生成源となりうる値の配列に，それらの値を含める．

ソースコード 3.15　関数呼出式（変換前）

```
1 exp_1(exp_2, ..., exp_n)
```

ソースコード 3.16　関数呼出式（変換後）

```
1 _PASS(exp_1^(exp_2^, ..., exp_n^), loc, [_REF['id(exp_1)'], ..., _REF['id(exp_n)']])
```

ただし，関数ポジションの式がオブジェクトのメンバ関数の参照式の場合（ソースコード 3.17, 3.20），メンバ関数を単体で評価した値には，その関数の呼び出し元のオブジェクトの情報が含まれておらず，ソースコード 3.16 の変換では関数呼出がうまくいかずエラーとなる．この問題に対しては 2 つの対処方法が考えられる．

1. 関数ポジションの式は変換しない（ソースコード 3.18, 3.21）
2. 関数ポジションの式と，関数呼出式全体を個別に変換する（ソースコード 3.19, 3.22）

対処方法 1 の場合，関数ポジションの式を評価して得られる関数値は追跡子付きの値とならず，もし関数呼出の結果新しい値が生成された場合に，その生成源の値に関数値を含めることができない．そのため，デバッグの精度が低下してしまう可能性がある．

対処方法 2 の場合，関数呼出式評価時に発生するすべての値が追跡子付きの値となる．しかし，関数ポジションの式（$tmp_1.prop$, $tmp_1[tmp_2]$）はプログラム実行時に 2 回評価される（関数ポジションの式単体の評価で 1 回，関数呼出式評価時の部分式としての関数ポジションの式の評価で 1 回）ため，これらの式が副作用を起こす場合（オブジェクトのゲッタからの関数呼出の場合などがある），プログラムのメインの処理に影響を与えてしまう．

実装では，本手法における，すべての値への追跡子付与による実行時情報収集の実現を優先し，対処方法 2 を採用することとした．また，関数呼出式評価時に発生するすべての値を追跡子付きの値とし，かつ，ゲッタの副作用にも対応するには，ゲッタ定義のコードと併せて副作用を考慮した変換を施す必要がある．

ソースコード 3.17　ドット記法によるメンバ関数呼出式（変換前）

```
1 exp_1.prop(exp_2, ..., exp_n)
```

ソースコード 3.18　ドット記法によるメンバ関数呼出式（変換後 1）

```
1 _PASS(exp_1~.prop(exp_2^, ..., exp_n^), loc', [_REF['id(exp_1)'], ..., _REF['id(exp_n)']])
```

ソースコード 3.19　ドット記法によるメンバ関数呼出式（変換後 2）

```
1 tmp_1 = exp_1~,
2 tmp_2 = _PASS(tmp_1.prop, loc, [_REF['id(exp_1)']]),
3 _PASS(tmp_1.prop(exp_2^, ..., exp_n^), loc', [_REF['id(tmp_2)'], _REF['id(exp_2)'], ...,
      _REF['id(exp_n)']])
```

ソースコード 3.20　ブラケット記法によるメンバ関数呼出式（変換前）

```
1 exp_1[exp_2](exp_3, ..., exp_n)
```

ソースコード 3.21　ブラケット記法によるメンバ関数呼出式（変換後 1）

```
1 _PASS(exp_1~[exp_2^](exp_3^, ..., exp_n^), loc', [_REF['id(exp_1)'], ..., _REF['id(exp_n)']])
```

ソースコード 3.22　ブラケット記法によるメンバ関数呼出式（変換後 2）

```
1 tmp_1 = exp_1~,
2 tmp_2 = exp_2~,
3 tmp_3 = _PASS(tmp_1[tmp_2], loc, [_REF['id(exp_1)'], _REF['id(exp_2)']]),
4 _PASS(tmp_1[tmp_2](exp_3^, ..., exp_n^), loc', [_REF['id(tmp_3)'], _REF['id(exp_3)'], ...,
      _REF['id(exp_n)']])
```

代入式

代入式の意味は，左辺の式を評価して得られた代入先の場所に対して，右辺の式を評価して得られた値を代入することである．そのため，左辺の式について，その式全体は変換対象とせず，部分式のみを再帰的に変換する．また，右辺の式は再帰的に変換する．また，右辺で代入された値が左辺に格納されたことを表す場所情報を取得するため，それぞれの loc は代入式の左辺の式（$name$, $exp_1.prop$, $exp_1[exp_2]$）の場所を表す．また，代入式全体の変換では，外の文脈に応じて適切な関数を使用する（変換例では_PASS 関数）．

ソースコード 3.23　代入式（変換前）

```
1 name = exp // 左辺が変数名の場合
2 exp_1.prop = exp_2 // 左辺がメンバ参照(ドット記法)の場合
3 exp_1[exp_2] = exp_3 // 左辺がメンバ参照(ブラケット記法)の場合
```

ソースコード 3.24　代入式（変換後）

```
1 _PASS(name = exp, loc, []) // 左辺が変数名の場合
2 _PASS(exp_1.prop = exp_2, loc, []) // 左辺がメンバ参照(ドット記法)の場合
3 _PASS(exp_1[exp_2] = exp_3, loc, []) // 左辺がメンバ参照(ブラケット記法)の場合
```

関数定義文

関数定義文では，この関数を呼び出した際に，実引数の値が仮引数に渡されたことを表す場所情報を取得するため，関数本体のブロックの先頭に，仮引数の式を_PASS 関数で変換している．ここで，$loc_1, \ldots, loc_n$ は，仮引数ポジションの仮引数名 $arg_1, \ldots, arg_n$ の場所を表す．また，$block$ は関数本体の処理を表しており，それらの処理中の式を適切に変換したものを $\widetilde{block}$ と記述している．

ソースコード 3.25　関数定義文（変換前）

```
1 function funcName(arg_1, ..., arg_n) {
2   block
3 }
```

ソースコード 3.26　関数定義文（変換後）

```
1 function funcName(arg_1, ..., arg_n) {
2   _PASS(arg_1, loc_1, []), ..., _PASS(arg_n, loc_n, []);
3   block~
4 }
```

ラムダ式

ラムダ式の場合も，関数定義文の場合と同様の変換を行うが，最後にラムダ式全体を外の文脈に応じて関数_PASS または_GETVAL に渡して変換する（変換例では_PASS 関数）．

ソースコード 3.27　ラムダ式（変換前）

```
1 (arg_1, ..., arg_n) => { block }
```

ソースコード 3.28　ラムダ式（変換後）

```
1 _PASS((arg_1, ..., arg_n) => {
2   _PASS(arg_1, loc_1, []), ..., _PASS(arg_n, loc_n, []);
3   block~
4 }, loc, [])
```

変数宣言文

変数宣言の場合も，代入式と同様，左辺の変換は行わなず，右辺の式のみを再帰的に変換する．そして，右辺で代入された値が左辺に格納されたことを表す場所情報を取得するため，変数宣言文評価後に，代入の左辺の変数が評価されるよう変換している．ここで *loc* は変数宣言文の左辺の変数名 *name* の場所を表す．

ソースコード 3.29　変数宣言文（変換前）

```
1 let name = exp;
```

ソースコード 3.30　変数宣言文（変換後）

```
1 let name = exp^
2 _PASS(name, loc, []);
```

また，変数の初期化をせず宣言のみの場合には，`undefined` で初期化されたとみなして変換を行う．実際, JavaScript では未初期化の変数を参照すると結果として値 `undefined` が返される．

ソースコード 3.31　未初期化変数宣言文（変換前）

```
1 let name;
```

ソースコード 3.32　未初期化変数宣言文（変換後）

```
1 let name = _PASS(undefined, loc,
      []);
2 _PASS(name, loc, []);
```

条件文（if，while，for）

条件文では，条件式の評価結果は JavaSctript の Bool 値であることが必要なため，各条件式 *test* は`_GETVAL` 関数によって変換する．また，各ブロックの変換の際には，生成源となりうる値の配列に，条件式の値（`_REF['`*id*(*test*)`']`）を含めるようにする．

ソースコード 3.33　条件文（変換前）

```
1  if(test) {
2    block_1
3  }
4  else {
5    block_2
6  }
7
8  while(test) {
9    block
10 }
11
12 for(exp_1; test; exp_2) {
13   block
14 }
```

ソースコード 3.34　条件文（変換後）

```
1  if(~test) {
2    ~block_1
3  }
4  else {
5    ~block_2
6  }
7
8  while(~test) {
9    ~block
10 }
11
12 for(^exp_1; ~test; ^exp_2) {
13   ~block
14 }
```

try-catch 文

try-catch 文では，catch の引数の式と各ブロック中の式を適切に変換する．また，try ブロックの先頭に，動的に付与するユーザ指定追跡子配列と，式評価後復帰用スタックの状態を，例外発生時復帰用のスタックに push してセーブするための関数_SAVE_DY_MKRS_EXCP を呼び出すようにする．catch ブロックの先頭では，保存した状態を，例外発生時復帰用のスタックからリストアするための関数_RESTORE_DY_MKRS_EXCP を呼び出すようにする．

ソースコード 3.35　try-catch 文（変換前）

```
1  try {
2    block_1
3  }
4  catch (exp) {
5    block_2
6  }
```

ソースコード 3.36　try-catch 文（変換後）

```
1  try {
2    _SAVE_DY_MKRS_EXCP();
3    ~block_1
4  }
5  catch (^exp) {
6    _RESTORE_DY_MKRS_EXCP();
7    ~block_2
8  }
```

3.3　クエリシステムの実装

クエリシステムの実装では，追跡子付与能力付きプログラムを実行後に得られた追跡子付きの値群を使用する．追跡子付きの値は JavaScript のオブジェクトとして表現してい

るため，クエリシステムも JavaScript で記述した．ユーザはクエリシステムへの問い合わせを行うが，これは JavaScript インタプリタに対してクエリ関数を呼び出すことで実現できる．そして，クエリ関数は追跡子情報を処理し，その情報を文字列情報，または，視覚的情報として提供する（図 3.5）．

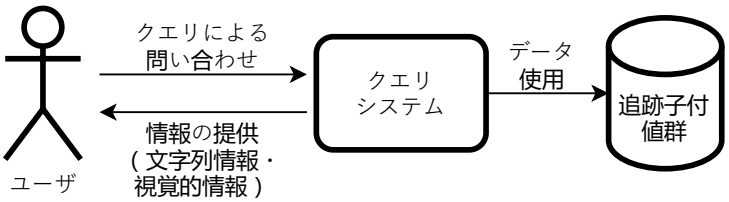

図 3.5　クエリシステムのイメージ

クエリシステムの実装では，実際の追跡子付きの値を処理する．追跡子付きの値オブジェクトは以下のプロパティを持つ．

- `id`：その値の ID
- `value`：追跡子付与対象の値
- `genIds`：その値の生成源の値の ID の配列
- `trace`：通過場所とその場所で付与されたユーザ指定追跡子文字列の配列．配列の要素のオブジェクトは以下のプロパティを持つ．
  - `loc`：通過場所を表すオブジェクト
  - `marker`：付与されたユーザ指定追跡子文字列の配列
  - `time`：通過時刻

例えばソースコード 3.37 のプログラム実行時に発生する値 3 の追跡子付きの値オブジェクトはソースコード 3.38 のような構造になる．また，このような実行時に発生した追跡子付きの値オブジェクトすべてを格納しておく大域コンテナは配列`_VALS` として実装した．

ソースコード 3.37　追跡子情報取得対象のプログラム

```
1 a = 1 + 2;
2 b = 2 * MK_EXP(a, 'refa');
```

ソースコード 3.38　値 3 の追跡子情報

```
1 {
2   id: 2, // この値のID
3   value: 3, // 追跡子付与対象の値
4   genIds: [0, 1], // 値の生成源の値の
        ID の配列．この追跡子の付与時刻は trace プロパティの配列の先頭要素の時刻(2)と同じ
```

```
5   trace: [ // 通過場所とその場所で付与されたユーザ指定追跡子文字列記録のための配列
6     {
7       loc: {start: [1, 4], end: [1, 9]}, // 1行目の右辺の式 1 + 2を通過したことを表
            す場所(1行 4列目から，1行 9列目の場所)
8       marker: [], // 通過時に付与されたユーザ指定の追跡子文字列
9       time: 2 // 通過時刻
10    },
11    {
12      loc: {start: [1, 0], end: [1, 1]}, // 1行目の左辺の変数
            a を通過したことを表す場所（1 行 0 列目から，1 行 1 列目の場所）
13      marker: [], // 通過時に付与されたユーザ指定の追跡子文字列
14      time: 3 // 通過時刻
15    },
16    {
17      loc: {start: [2, 11], end: [2, 12]}, // 2行目の右辺の変数
            a を通過したことを表す場所（2 行 11 列目から，2 行 12 列目の場所）
18      marker: ['refa'], // 通過時に付与されたユーザ指定の追跡子文字列
19      time: 5 // 通過時刻
20    },
21  ]
22 }
```

本実装では，2.2 節で示したクエリのうち，以下のもののみを実装している．

- JavaScript で記述された述語から，その述語を満たす値のみを取り出すクエリ
- 値からその値が通過したプログラム中の場所を記憶する追跡子を取り出すクエリ（クエリ 1）
- 値からその値の生成源の値を記憶する追跡子を取り出すクエリ（クエリ 2）
- プログラム中の場所から，その追跡子を持つ値のみを取り出すクエリ（クエリ 4 を部分的に実装）
- ユーザ指定の追跡子文字列から，その追跡子を持つ/持たない値のみを取り出すクエリ（クエリ 6）
- ユーザ指定の追跡子文字列の付与のされ方に関する条件を満たす値のみを取り出すクエリ（クエリ 7）
- 値の集合に対する集合演算クエリ（クエリ 8）

まず，JavaScript で記述された述語から，その述語を満たす値のみを取り出すクエリは，関数 `filterValue` として実装した．引数に与える述語は，JavaScript の値を引数にとり，真偽値を返す．ソースコード 3.39 は，ソースコード 3.37 のプログラムに対して，クエリ `filterValue` を使用した例である．このクエリの実装では，`_VALS` に含まれる追

跡子付きの値オブジェクト v の中から，filterValue の述語に v.value を適用した結果が true となるような v のみを要素とする配列を返せばよい．また，クエリの結果の出力として，ユーザに必要十分な情報を提供するため，追跡子付きの値オブジェクトの文字列表示では，ID と値のみを表示するように設定している．

ソースコード 3.39　クエリ filterValue の使用例

```
1 > filterValue(v => v == 3) // 実行時に発生した個々の値 3をすべて取得
2 [ {id: 2, value: 3} ]
```

クエリ 1 は，関数 showTrace として実装した．この関数は，引数として値の ID を渡すと，その ID の値が通過したプログラム中の場所を，エディタ上（図 3.1，要素 A）に描画する（図 3.2）．このクエリの実装では，まず引数として与えられた ID をプロパティ id に持つ追跡子情報オブジェクトから trace プロパティの配列を取り出す．そして，取り出した配列の各要素のオブジェクトの loc プロパティの情報を利用して，エディタ上での描画位置を計算し，値の通過場所の描画を行う．ソースコード 3.40 は，ソースコード 3.37 のプログラムに対して，クエリ showTrace を使用した例である．この実行によって，クエリコンソール上ではクエリが正しく実行されたことを表す文字列'OK' が返され，プログラム上に値の通過経路が描画される（図 3.6）．図 3.6 で描画された通過経路から，値 3 は 1 行目の右辺の式 1 + 2 から発生し，その後 1 行目の左辺の変数 a に格納され，最後に 2 行目の変数 a から取り出されたことが分かる．

ソースコード 3.40　クエリ showTrace の使用例

```
1 > showTrace(2) // 値 3のID
2 'OK' // エディタ上に値 3の通過経路が描画される
```

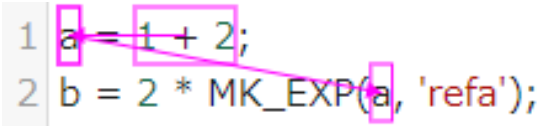

図 3.6　エディタ上に描画された経路

クエリ 2 は，関数 findGen として実装した．この関数は，引数として値の ID を渡すと，その ID の値の生成源となった値の情報を，クエリコンソール上に文字列として提供する．ソースコード 3.41 は，ソースコード 3.37 のプログラムに対して，クエリ findGen を使用した例である．このクエリの実装では，まず引数として与えられた ID をプロパティ id に持つ追跡子情報オブジェクトから genIds プロパティの配列を取り出す．そして，その配列の各要素の数値を ID として持つ値の情報を取り出して提供する．

ソースコード 3.41　クエリ findGen の使用例

```
> findGen(2) // ID が 2 の値は 3 で，この値は式 1 + 2の結果
[
  {id: 0, value: 1},
  {id: 1, value: 2}
]
```

クエリ 4 は，関数 findValue として部分的に実装した（指定した場所を記憶する追跡子を持たない値を取り出すクエリは実装していない）．この関数は，エディタ上のプログラムの一部を選択した上で実行する．実行すると，エディタ上のプログラムで選択された部分に相当する式の場所を通過した値の情報を，クエリコンソール上で文字列として提供する．このクエリの実装では，エディタ上のプログラムで選択された部分の場所情報（選択開始/終了場所の行・列数）を取得し，その場所情報と同じ場所を表す情報を持っている追跡子情報の値の情報を取り出して提供する．ソースコード 3.42 は，ソースコード 3.37 のプログラムに対して，図 3.7 のようにエディタ上で変数 a を選択した後，クエリ findValue を使用した例である．

```
1 a = 1 + 2;
2 b = 2 * MK_EXP(a, 'refa');
```

図 3.7　エディタ上で選択された変数 a

ソースコード 3.42　クエリ findValue の使用例

```
> findValue() // エディタ上で変数a を選択した状態で実行
[ {id: 2, value: 3} ]
```

クエリ 6 は，それぞれ，関数 hasMkr，nhasMkr として実装した．この関数は，引数として文字列を渡すと，その文字列をユーザ指定の追跡子文字列として持つ値の情報を，クエリコンソール上に文字列として提供する．このクエリの実装では，引数として与えられた文字列を，ユーザ指定の追跡子文字列として持つ追跡子情報の値の情報を提供する．ソースコード 3.43 は，ソースコード 3.37 のプログラムに対して，クエリ hasMkr，nhasMkr を使用した例である．ここで，LOG は，取得した値全体の集合を表す．

ソースコード 3.43　クエリ hasMkr，nhasMkr の使用例

```
> LOG.hasMkr('refa') // すべての値の中から,ユーザ指定の追跡子文字列'refa'を持つ値のみを取得
[ {id: 2, value: 3} ]
> LOG.nhasMkr('refa') // すべての値の中から,ユーザ指定の追跡子文字列'refa'を持たない値のみを取得
```

```
[
  {id: 0, value: 1},
  {id: 1, value: 2}
]
```

クエリ 7 は，*mkrEveryTime*, *mkrSomeTime* を，それぞれ関数 mkrEveryTime，mkrSomeTime として実装した．この関数は 3 つの引数をとり，mkrEveryTime(mkr_1，**R**，mkr_2) のようにして実行する．それぞれの引数は文字列であるが，mkr_1, mkr_2 には，ユーザ指定の追跡子文字列を渡し，**R** には，mkr_1, mkr_2 に対する論理演算 AND, OR, XOR, NAND, NOR, XNOR, $\Rightarrow$（ならば），$\not\Rightarrow$（ならば，の否定）のそれぞれを表す文字列'AND', 'OR', 'XOR', 'NAND', 'NOR', 'XNOR', '->', '!->' を渡すことができる．ソースコード 3.45 は，ソースコード 3.44 のプログラム（2.2.3 節で示したプログラムを再掲）に対して，クエリ mkrEveryTime，mkrSomeTime を使用した例である．ここで，LOG は，取得した値全体の集合を表す．

ソースコード 3.44　ユーザ指定追跡子文字列付与を使用したプログラム（再掲）

```
function f(x) {
  return MARK(x, 'in_f'); // バリエーション 1方式での追跡子文字列付与を指示
}
let a = 0;
MARK_DY(f(a), 'call_f'); // バリエーション 3方式での追跡子文字列付与を指示
MARK(a, 'out_f'); // バリエーション 1方式での追跡子文字列付与を指示
```

ソースコード 3.45　クエリ mkrEveryTime，mkrSomeTime の使用例

```
> LOG.mkrEveryTime('in_f', '->', 'call_f') // すべての付与時刻で，'in_f'が付与されているなら'call_f'も付与されているような値を取得
[ {id: 0, value: 0} ]
> LOG.mkrSomeTime('in_f', 'AND', 'call_f') // 'in_f'と'call_f'が同時に付与されている時刻があるような値を取得
[ {id: 0, value: 0} ]
> LOG.mkrEveryTime('call_f', 'NAND', 'out_f') // 'call_f'と'out_f'が同時刻に付与されることがないような値を取得
[ {id: 0, value: 0} ]
```

findGen，findValue は値の集合を返すクエリであり，hasMkr，nhasMkr，mkrEveryTime，mkrSomeTime は，値の集合を受け取り，値の集合を返すクエリである．これらのクエリの結果の集合に関する集合演算クエリ（クエリ 8）は，メソッドチェーンとして実現した．例えば，ある場所を通過した値，かつ，ある追跡子文字列 *mkr* を持つ値の集合を求めるクエリは，findValue().hasMkr(*mkr*) と記述できる．ま

た，ある追跡子文字列 mkr_1, mkr_2 の少なくとも 1 つを持つ値の集合を求めるクエリは，`LOG.hasMkr(`mkr_1`).Union(hasMkr(`mkr_2`))` と記述できる．ここで，`Union` は，その関数の呼出元の集合と，引数の集合の和集合を返す関数である．

第 4 章

ツールの利用例

この章では，実装したツールを利用して，実際のプログラムに対してデバッグを行う例を示す．

4.1 意図しない挙動のデバッグ

典型的なバグとして，計算結果を出力するプログラムで，意図しない値が出力される場合があるが，本節ではそのようなプログラムのデバッグや調査の例を示す．

4.1.1 値の生成源を遡るデバッグ

ソースコード 4.1 は，数値の配列 a の要素の総和を求めて出力するプログラムであるが，このプログラムを実行すると，意図しない値 NaN が出力される（正しい出力は 8）．

ソースコード 4.1 配列の要素の総和を求めるプログラム（バグあり）

```
1 let a = [3, 1, 4];
2 let sum = 0;
3 for (let i = 0; i <= a.length; i = i + 1) {
4   sum =
5     sum + a[i];
6 }
7 console.log(sum); // NaN を出力
```

このプログラムのデバッグでは，まず，7 行目の式 sum の場所を通過した値のみを取得することにした．このために，まずプログラム中の 7 行目の式 sum を選択し（図 4.1），クエリ findValue を実行する（図 4.2）．この結果として，値 NaN とその ID 情報{id:24, value:NaN}が得られる．

次に，出力された値 NaN の生成源の値を調べる．そのために，値 NaN の ID 24 を使用して，クエリ findGen(24) を実行する．これにより図 4.3 の結果が得られる．これらの値の通過場所を調べるためには，クエリ showTrace(18)，showTrace(23)，showTrace(22) をそれぞれ実行する．すると，それぞれの値の通過経路は，図 4.4，4.5，4.6 のようにプ

```
1 let a = [3, 1, 4];
2 let sum = 0;
3 for (let i = 0; i <= a.length; i = i + 1) {
4   sum =
5     sum + a[i];
6 }
7 console.log(sum); // NaN
```

図 4.1　エディタ上で選択された変数 sum

```
> findValue()
[ { id: 24, value: NaN } ]
>
```

図 4.2　クエリ findValue の実行とその結果（変数 sum を通過した値）

ログラム字面上に描画される．これにより，値 8 は 5 行目の演算 sum + a[i] によって発生し，その後 4 行目の変数 sum に格納され，for ループを一周した後に，再度 5 行目の演算の右辺の変数 sum から取り出されたことが分かる．値 undefined は，5 行目の演算の右辺の式 a[i] を通過したことが分かる．値 true は，3 行目の for 文内の条件式 i <= a.length を通過したことが分かる．つまり，値 NaN は，5 行目の演算 sum + a[i] として，8 + undefined が行われたことによって生成されたということが分かる．

```
> findGen(24)
[ { id: 18, value: 8 },
  { id: 23, value: undefined },
  { id: 22, value: true } ]
>
```

図 4.3　クエリ findGen(24) の実行とその結果（値 NaN の生成源の値）

```
1 let a = [3, 1, 4];
2 let sum = 0;
3 for (let i = 0; i <= a.length; i = i + 1) {
4   sum =
5     sum + a[i];
6 }
7 console.log(sum); // NaN
```

図 4.4　クエリ showTrace(18) の実行結果（値 8 の通過場所）

```
1 let a = [3, 1, 4];
2 let sum = 0;
3 for (let i = 0; i <= a.length; i = i + 1) {
4   sum =
5     sum + a[i];
6 }
7 console.log(sum); // NaN
```

図 4.5　クエリ showTrace(23) の実行結果（値 undefined の通過場所）

同様に，クエリ fundGen(23) を実行することにより，値 undefined の生成源の値が得られる（図 4.7）．これらの値の通過場所は，showTrace(3)，showTrace(20) によって調べられる（ID 22 の値 true の通過場所は確認済み）．その実行結果はそれぞれ，図 4.8，4.9 のようになる．これより，それぞれ，5 行目の配列 a の値，その配列の添字の値であることが分かる．よって，値 undefined は，配列外参照 a[3] によって意図せず発生し

```
1 let a = [3, 1, 4];
2 let sum = 0;
3 for (let i = 0; i <= a.length; i = i + 1) {
4   sum =
5     sum + a[i];
6 }
7 console.log(sum); // NaN
```

図 4.6　クエリ `showTrace(22)` の実行結果（値 `true` の通過場所）

たものであることが分かる．ここまで分かれば，3 行目の for 文の条件式 `i <= a.length` がバグの原因であることに気が付くと予想される（正しくは `i < a.length`）．

```
> findGen(23)
[ { id: 3,
    value: [ { id: 0, value: 3 }, { id: 1, value: 1 }, { id: 2, value: 4 } ] },
  { id: 20, value: 3 },
  { id: 22, value: true } ]
>
```

図 4.7　クエリ `findGen(23)` の実行結果（値 `undefined` の生成源の値）

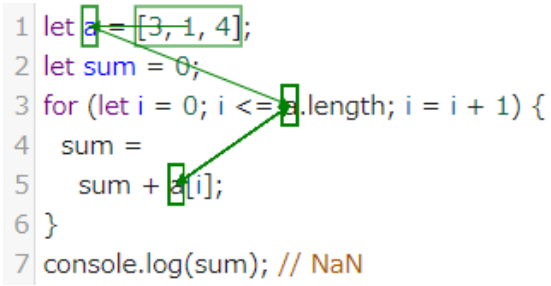

図 4.8　クエリ `showTrace(3)` の実行結果（値 `[3, 1, 4]` の通過場所）

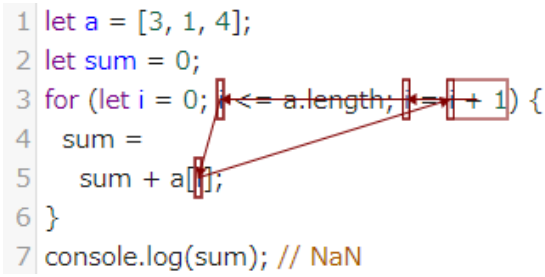

図 4.9　クエリ `showTrace(20)` の実行結果（値 3 の通過場所）

このようなデバッグ過程は，動的プログラムスライシング技術 [6] を利用したデバッグ手法であるアルゴリズミックデバッギング [9] に類似している．具体的な類似点については，6 章で議論する．

4.1.2　値の流れの調査によるプログラムの挙動の調査

ソースコード 4.3 は，クイックソートを行うプログラムで，文科省の情報 I の教員用研修教材 [5] に掲載されているプログラムを再現（原文では Python で記述されていたものを JavaScript で記述した）したものである．

ソースコード 4.2　クイックソートプログラム（バグあり）

```
1  function quicksort(a, start, end) {
2    let m = Math.floor((start + end) / 2);
3    let i = start; let j = end;
4    while (i < j) {
5      while (a[i] < a[m]) i = i + 1;
6      while (a[j] > a[m]) j = j - 1;
7      if (i >= j) break;
8      const temp = a[i];
9      a[i] = a[j]; // 右の値を左に移動
10     a[j] = temp; // 左の値を右に移動
11     if (i == m) { m = j; }
12     else if (j == m) { m = i; }
13     i = i + 1; j = j - 1;
14   }
15   if (start < i - 1) quicksort(a, start, m - 1);
16   if (end > j + 1) quicksort(a, m + 1, end);
17 }
18 let a = [7, 22, 11, 34, 17, 52, 26, 13];
19 console.log(`before: ${a}`);
20 quicksort(a, 0, a.length - 1);
21 console.log(`after: ${a}`);
```

このプログラムを実行すると，図 4.10 のように，ソート前後での配列の中身が出力されるが，ソート後の配列において，13 と 11，52 と 34 が正しくソートされていないことが分かる．また，このバグの原因を特定することは容易ではないことが報告されている [4]．そこで，13 と 11 に注目し，ソースコード 4.3 の実行時の値の不審な動きの調査を試みる．

```
'before: 7,22,11,34,17,52,26,13'
'after: 7,13,11,17,22,26,52,34'
```

図 4.10　ソースコード 4.3 の実行結果

まず，値 11 の通過経路をクエリ showTrace によって表示すると図 4.11 のようになる．この結果から，値 11 は 5 行目の条件式で参照されるのみであり，クイックソートの処理中配列内を一度も移動せず留まっていたことが分かる．

また，値 13 の通過経路をクエリ showTrace によって調べると図 4.12 のようになる．この結果から，値 13 は，9 行目の配列内で要素を移動させる処理を通過しており，配列内を移動していることが分かる．ただし，この結果のみから，その移動が正しいかどうかを判断することはできない．

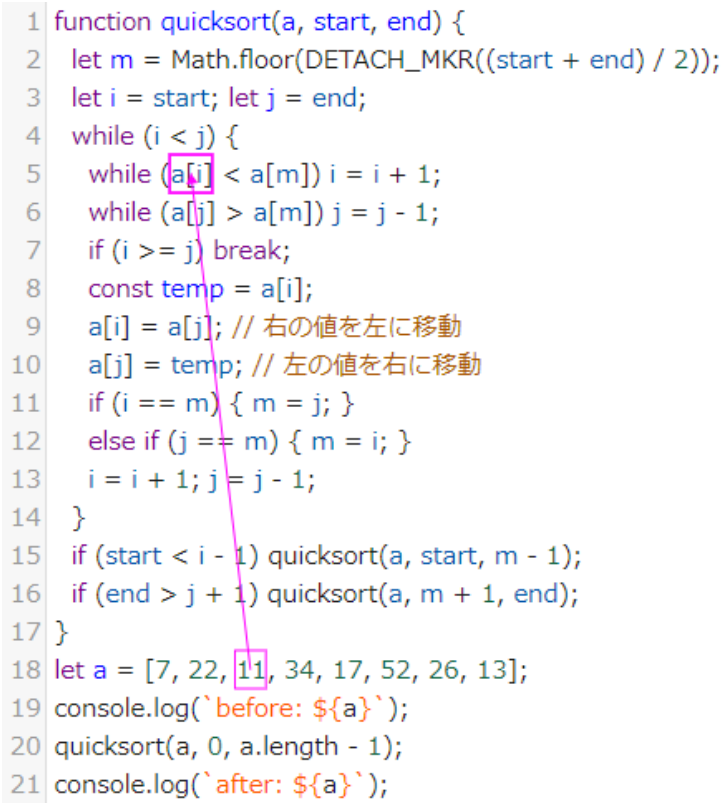

図 4.11　値 11 の通過経路

次に，値 13 は配列の 1 番目に移動したが，なぜその場所に移動してきたのかを調べるため，添字の値 1 の通過経路を調べる．そのために，まず，9 行目の左辺の添字で使用されている変数 i を範囲選択し，クエリ findValue を実行する．これにより，図 4.13 のような結果が得られ，ID が 71 の値 1 が，調べたい値である．そして，この値 1 の通過経路をクエリ showTrace によって調べると図 4.14 のようになる．この結果から，添字の値 1 は 9 行目で使用された後，12 行目で変数 m に渡され，15 行目と 16 行目でクイックソートの両方の再帰に渡されている．そして，クイックソートの再帰は，添字の値 1 の前後で分割した配列に対して行われており，添字の 1 番目の値 13 はソートの対象外となってしまっていることが分かる．これは，クイックソートが正しく行われない原因である．

このようにして，値の流れを調査するだけで，プログラムの挙動の調査を進めることができる．

```
1  function quicksort(a, start, end) {
2    let m = Math.floor(DETACH_MKR((start + end) / 2));
3    let i = start; let j = end;
4    while (i < j) {
5      while (a[i] < a[m]) i = i + 1;
6      while (a[j] > a[m]) j = j - 1;
7      if (i >= j) break;
8      const temp = a[i];
9      a[i] = a[j]; // 右の値を左に移動
10     a[j] = temp; // 左の値を右に移動
11     if (i == m) { m = j; }
12     else if (j == m) { m = i; }
13     i = i + 1; j = j - 1;
14   }
15   if (start < i - 1) quicksort(a, start, m - 1);
16   if (end > j + 1) quicksort(a, m + 1, end);
17 }
18 let a = [7, 22, 11, 34, 17, 52, 26, 13];
19 console.log(`before: ${a}`);
20 quicksort(a, 0, a.length - 1);
21 console.log(`after: ${a}`);
```

図 4.12　値 13 の通過経路

```
> findValue()
[ { id: 33, value: 3 },
  { id: 45, value: 5 },
  { id: 71, value: 1 },
  { id: 126, value: 3 } ]
```

図 4.13　添え字の変数 i を通過した値一覧

```
1  function quicksort(a, start, end) {
2    let m = Math.floor(DETACH_MKR((start + end) / 2));
3    let i = start; let j = end;
4    while (i < j) {
5      while (a[i] < a[m]) i = i + 1;
6      while (a[j] > a[m]) j = j - 1;
7      if (i >= j) break;
8      const temp = a[i];
9      a[i] = a[j]; // 右の値を左に移動
10     a[j] = temp; // 左の値を右に移動
11     if (i == m) { m = j; }
12     else if (j == m) { m = i; }
13     i = i + 1; j = j - 1;
14   }
15   if (start < i - 1) quicksort(a, start, m - 1);
16   if (end > j + 1) quicksort(a, m + 1, end);
17 }
18 let a = [7, 22, 11, 34, 17, 52, 26, 13];
19 console.log(`before: ${a}`);
20 quicksort(a, 0, a.length - 1);
21 console.log(`after: ${a}`);
```

図 4.14　値 1 の通過経路

4.2 関心のない値の除外

ソースコード 4.1 のデバッグで発見した値 undefined は意図せず発生したものだったが，ユーザが意図してプログラム中に undefined を記述する場合もある．ソースコード 4.3 は，配列として表現された 2 次元座標の値を要素として持つ配列 a の要素の x 座標の総和を求めるプログラムである．このプログラムを実行すると例外が発生し，以下のようなメッセージが報告される．

TypeError: Cannot read properties of undefined (reading '0')

このメッセージから，実行時に発生した値 undefined が原因となって例外が発生したことが分かる．そこで，実行時に発生したすべての値の中から，クエリ filterValue(v => v === undefined) によって undefind のみを絞り込み，原因の候補を探る．その結果は図 4.15 のようになる．

ソースコード 4.3 2 次元座標の配列の要素の x 座標の総和を求めるプログラム（バグあり）

```
1  function makePoint(x, y) {
2    return ['Point', x, y];
3  }
4  function getX(p) {
5    if(p[0] !== 'Point') {
6      return undefined;
7    }
8    else {
9      return p[1];
10   }
11 }
12 let a = [makePoint(3, 1), makePoint(4, 1), makePoint(5, 9)];
13 let sum = 0;
14 for (let i = 0; i <= a.length; i = i + 1) {
15   let x = getX(a[i]);
16   if(x !== undefined) {
17     sum =
18       sum + x;
19   }
20 }
21 console.log(sum); // 12が出力されることを期待
```

得られた値 undefined には，ユーザが意図して記述したものも含まれる．それらの undefined を抽出結果から除外し，より関心のある値のみを取得するために，ユーザ指定の追跡子とクエリを利用できる．ユーザは意図して記述した undefined を通

```
> filterValue(v => v === undefined)
[ { id: 25, value: undefined },
  { id: 37, value: undefined },
  { id: 49, value: undefined },
  { id: 57, value: undefined } ]
>
```

図 4.15　クエリ `filterValue(v => v === undefined)` の実行結果（値 `undefined` の抽出）

過する値に対して，ユーザ指定の追跡子文字列`'intended'` を付与するように追跡子付与ポイントを設定する（図 4.16）．そして，`'intended'` を持つ値を除外するクエリ `nhasMkr('intended')` を実行する．この結果は図 4.17 のようになる．

```
1  function makePoint(x, y) {
2    return ['Point', x, y];
3  }
4  function getX(p) {
5    if(p[0] !== 'Point') {
6      return MK_EXP(undefined, 'intended');
7    }
8    else {
9      return p[1];
10   }
11 }
12 let a = [makePoint(3, 1), makePoint(4, 1), makePoint(5, 9)];
13 let sum = 0;
14 for (let i = 0; i <= a.length; i = i + 1) {
15   let x = getX(a[i]);
16   if(x !== MK_EXP(undefined, 'intended')) {
17     sum =
18       sum + x;
19   }
20 }
21 console.log(sum);
```

図 4.16　意図して記述した `undefined` への追跡子付与ポイントの設定（選択箇所）

この結果から，ID 57 の値 `undefined` のみが意図せず発生したものであることが分かる．そして，この値を起点として，4.1 節のデバッグを開始できる．

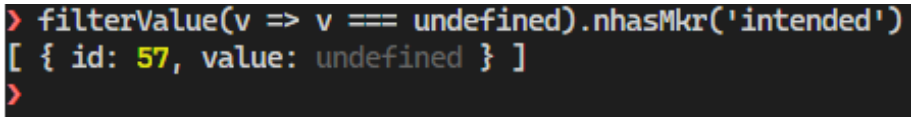

図 4.17　値 undefined の集合に対するクエリ nhasMkr('intended') の実行結果（意図せず発生した値 undefined の抽出）

4.3　プログラムの性質の検査

ユーザ指定の追跡子文字列とクエリは様々な用途に利用できる．特に，デバッグ対象のプログラムに，バグを発生しうるプログラミングパターンが含まれていないかどうかを検査するために利用できる．

例 1：タグチェックせずにオブジェクトの要素アクセスがされていないことの保証

Scheme [3] などの言語では，ユーザ定義のオブジェクトを cons セルによるリスト構造として表現することがある．このとき，リストの先頭にそのオブジェクトの型を表すシンボルをタグとして格納しておき，使用時にはタグをチェックしてオブジェクトの型を識別し，そのオブジェクトの種類に応じて格納されている要素を取り出す（ソースコード 4.4）．

ソースコード 4.4　point オブジェクトをリスト構造として表現した Scheme プログラム

```
1  ;; point オブジェクトのコンストラクタ
2  (define (make-point x y) (cons 'point (cons x y)))
3
4  ;; point オブジェクトの x のゲッター関数
5  (define
6    (point-get-x p)
7    (if
8      (eq? (car p) 'point) ;; 引数に渡された値のタグチェック
9      (cadr p)
10     (error "unexpected value p")))
11
12 ;; main 処理
13 (define p (make-point 2 3)) ;; point オブジェクトの生成
14 (point-get-x p) ;; x のゲッター関数を使用した（タグチェックをした上での）アクセス
```

しかし，オブジェクトの実体はリストデータのため，ソースコード 4.5 のように，リストデータのアクセサを使用してタグチェックをせずにオブジェクトの要素にアクセスする

こともできてしまう．このような記述は直ちにバグを発生させるわけではないが，開発を進めていく中で将来的にバグの原因となりうるものである．

ソースコード 4.5　タグチェックをしないオブジェクトの要素へのアクセス

```
1 (define p (make-point 2 3)) ;; point オブジェクトの生成
2 (cadr p) ;; タグチェックをせずにアクセス
```

このようなプログラミングパターンを，ユーザ指定の追跡子文字列とクエリによって検知することを試みる．まず，ソースコード 4.4 を JavaScript で記述したプログラムをソースコード 4.6 に示す．

ソースコード 4.6　point オブジェクトをリスト構造として表現した JavaScript プログラム

```
1  function makePoint(x, y) { // point オブジェクトのコンストラクタ
2    return ['point', x, y];
3  }
4  function getX(p) { // point オブジェクトの x のゲッター関数
5    if(p[0] !== 'point') {
6      return undefined;
7    }
8    else {
9      return p[1];
10   }
11 }
12 // main 処理
13 let p1 = makePoint(2, 3);
14 let p1_x = getX(p1); // OK
15 let p2 = makePoint(4, 5);
16 let p2_x = p2[1]; // NG
```

このプログラムに対して，ユーザ指定の追跡子文字列付与ポイントを設置する．まず，検査対象の値は，point オブジェクト全体のため，point オブジェクトのコンストラクタ関数内 2 行目の配列式を通過した値に，その値が point オブジェクトの値であることを表現するための追跡文字列'point' を付与するよう指定する．また，9 行目のタグチェック済みのブロックに対して，実行時にそのブロック内で評価された値に，その値が point オブジェクトであることを確認済みであることを表現するための追跡子文字列'checked' を付与するよう指定する．また，配列の要素参照を使用して point オブジェクトにアクセスしている箇所を通った値に対して，配列の要素参照が行われたことを表現するための追跡子文字列'get' を付与するよう指定する．これらの指定を行ったプログラムはソースコード 4.7 のようになる．

ソースコード 4.7　point オブジェクトをリスト構造として表現した JavaScript プログラム（追跡子付与ポイント設置）

```
1  function makePoint(x, y) { // point オブジェクトのコンストラクタ
2    return MK_EXP(['point', x, y], 'point');
3  }
4  function getX(p) { // point オブジェクトの x のゲッター関数
5    if(p[0] !== 'point') {
6      return undefined;
7    }
8    else {
9      MK_BLK_DY('checked');
10     return MK_EXP_ST(p[1], 'get');
11   }
12 }
13 // main 処理
14 let p1 = makePoint(2, 3);
15 let p1_x = getX(p1); // OK
16 let p2 = makePoint(4, 5);
17 let p2_x = MK_EXP_ST(p2[1], 'get'); // NG
```

これに対して，「point オブジェクトのうち，タグチェックされていないのに，配列参照によって要素アクセスがされている値」は，以下のクエリによって取得できる．

```
LOG.hasMkr('point').nhasMkr('checked').hasMkr('get')
```

このクエリを実際にソースコード 4.7 に対して実行した結果を図 4.18 に示す．これはソースコード 4.7 の 16 行目で変数 p2 に格納されたオブジェクトであり，良くないプログラミングパターンで使用された値を特定することができている．

```
> LOG.hasMkr('point').nhasMkr('checked').hasMkr('get')
[ { id: 15,
    value:
     [ { id: 14, value: 'point' },
       { id: 12, value: 4 },
       { id: 13, value: 5 } ] } ]
>
```

図 4.18　タグチェックせずに要素アクセスが行われた値を特定するクエリの実行結果

しかし，前述のクエリでは，「ある時はタグチェックして要素アクセスされ，別のある時はタグチェックをせずに要素アクセスをされた値」を特定することができない．この値を特定するクエリを記述するためには，hasMkr，nhasMkr だけでは表現力が足りない．このような場合には，クエリ mkrEveryTime，mkeSomeTime が有効である．期待するプログラミングパターンは，「タグチェックされたスコープ内の場合にのみ，要素アクセ

スがされている」ことであり，これはクエリ mkrEveryTime('get', '->', 'check') と表せる．今回検知したい値は，この条件を否定した条件を満たす値であり，これはクエリ mkrSomeTime('get', '!->', 'check') と表せる．よって，以下のクエリによって，良くないプログラミングパターンで使用されたすべての値を漏れなく取得することができる．

```
LOG.hasMkr('point').mkrSomeTime('get', '!->', 'check')
```

例 2：オブジェクトの内容が書き換えられないことの保証

JavaScript では，const 修飾子を使用して変数宣言をすることで，その変数に格納された値，またはオブジェクトへの参照が書き換え不可能であることを保証できる．しかし，参照先のオブジェクトの内容が書き換えられないことは保証されない（ソースコード 4.8）．

ソースコード 4.8　オブジェクトの内容変更を行うプログラム

```
1 function makePoint(x, y) {
2   return {x: x, y: y};
3 }
4 const p = makePoint(2, 3); // オブジェクトの中身も書き換えられてほしくない
5 p = makePoint(1, 1); // 実行時エラー
6 p.x = 0; // オブジェクトの中身の書き換え(実行可能)
```

参照先のオブジェクトの内容が書き換えられないことを保証するために，ユーザ指定の追跡子文字列をクエリを利用できる．まず，中身の書き換えを禁止したい point オブジェクトに対して，その内容が書き換えられてほしくない（読み取り専用である）ことを表現するための追跡子文字列'readonly' を付与するよう指示する．また，プログラム中でオブジェクトのメンバの書き換えを行っている式の，メンバの呼び出し元の式に対して，その内容が書き換えられることを表現するための追跡子文字列'mutate' を付与するよう指示する．これらの指定を行ったプログラムはソースコード 4.9 のようになる（実行時エラーとなる行はコメントアウトしておく）．

ソースコード 4.9　オブジェクトの内容変更を行うプログラム（追跡子付与ポイント設置）

```
1 function makePoint(x, y) {
2   return {x: x, y: y};
3 }
4 const p = MK_EXP(makePoint(2, 3), 'readonly'); // オブジェクトの中身も書き換えられ
      てほしくない
5 // p = makePoint(1, 1); // 実行時エラー
6 MK_EXP(p, 'mutate').x = 0; // オブジェクトの中身の書き換え(実行可能)
```

このプログラムを実行することで得られた追跡子付きの値群の中から，その内容が書き換えられてほしくないのに書き換えられてしまっている値は，以下のクエリによって取得できる．

```
LOG.hasMkr('readonly').hasMkr('mutate')
```

このクエリを実際にソースコード 4.9 に対して実行した結果を図 4.19 に示す．これは 4 行目で生成されたオブジェクトであり，意図せずオブジェクトの中身が書き換えられた値を検知できている．

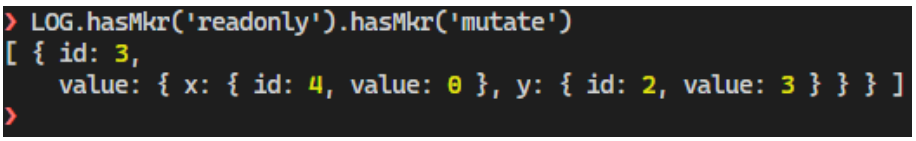

図 4.19　意図せず内容が書き換えられてしまっている値を特定するクエリの実行結果

第 5 章

問題点と課題

5.1 収集データ量の巨大化への対処（データ収集方法の工夫）

本手法では，プログラム実行時に現れる個々の値をすべて区別し，それらに追跡子を付与している．そのため，プログラムの規模に関わらず，ループや再帰があるプログラムや，長時間動作し続けるプログラムでは，収集する追跡子付きの値の量が膨大になってしまうことがある．例えば，ソースコード 5.1 は再帰を用いてフィボナッチ数列の第 20 項目を求めるプログラムであるが，現状の本手法において，このプログラム実行後に得られる値の数は 10 万個を超える．

ソースコード 5.1 フィボナッチ数列の第 20 項目を求めるプログラム

```
1 function fib(n) {
2   if(n == 1 || n == 2) {
3     return 1;
4   }
5   return fib(n - 1) + fib(n - 2);
6 }
7 console.log(fib(20)); // 6765を出力
```

このような収集データ量の巨大化への対処として，kCFA（kth-order Control-Flow-Analysis）[12] の考え方を利用できるのではないかと考えている．kCFA は，高階言語で記述されたプログラム実行時に起こりうる制御フローの候補の集合を静的解析によって得る手法である．特に，0CFA では，ある関数 f を複数回呼び出した際に，関数 f 中の同一の変数に束縛されうる個々の値を区別しない．この考え方に基づくと，本手法における値への追跡子付与ルールは以下のように変更できる．

0CFA に基づいた追跡子付与ルール

プログラム中のある場所で新しい値 v_{new} が生成された場合，

- 過去にその場所から生成された値 v_{old} がすでに追跡子付きの値として存在していたならば，v_{new} は追跡子付きの値とせず，v_{new} に付与すべきだった追跡子は v_{old} に付与する
- v_{old} が存在しない場合，v_{new} を追跡子付きの値とし，付与すべき追跡子を付与する

これにより，ループや再帰があるプログラムに対しても，収集する追跡子付きの値データの量は，プログラムの字面のサイズに比例した量となり，収集データ量を削減することができる．

また，さらなるデータ量削減の方法として，型が同じ値をすべて同一とみなすという考えもある．この考え方に基づくと，本手法における値への追跡子付与ルールは以下のように変更できる．

同一型の値を同一とみなす考えに基づいた追跡子付与ルール

プログラム中で新しい値 v_{new} が生成された場合，

- v_{new} の型と同じ型の値 v_{old} がすでに追跡子付きの値として存在していたならば，v_{new} は追跡子付きの値とせず，v_{new} に付与すべきだった追跡子は v_{old} に付与する
- v_{old} が存在しない場合，v_{new} を追跡子付きの値とし，付与すべき追跡子を付与する

しかし，これらの収集データ量の削減を行った情報に対しても，デバッグなどに有用な情報が得られるかについては議論の余地があり，これは今後の課題とする．

5.2 追跡子情報取得範囲の限界

本手法では，プログラム変換によって，値への追跡子付与を実現している．そのため，プログラム変換が行えないものに関しては，値への追跡子付与ができない．例えば，実装対象の言語が用意している組み込み関数や，外部ライブラリの API 関数などがこれに該当する．そのため，例えば，外部 API 関数を使用した関数呼出式 `API_CALC(1)` に対して，この評価結果の値の生成源の値は関数 `API_CALC` の値と実引数の値 `1` とみなされ，API 関数内の処理によって発生する値は追跡子の範囲外となる．そのため，本手法は，ユーザ自身が記述したプログラムのデバッグのために有用な手法であると言える．

5.3 ユーザ指定追跡子付与ポイントの自動設定

4 章では，ユーザが意図して記述した値 undefined や，配列の要素参照箇所などにユーザ指定追跡子付与ポイントを手動で設定した．しかし，ユーザが記述したプログラムの式に関する情報は，そのプログラムを静的に解析することで取得可能である．そのため，ユーザ指定追跡子付与ポイントの自動設定が可能である．この機能は，ユーザ指定追跡子を使用したデバッグをより効率的に行うために重要であると考えられるため，今後ツールに実装予定である．この機能は以下のような手順で使用できることを予定している．

1. ユーザは式の形状（シンボル undefined，配列参照式 exp_1[exp_2] など）と，その式を通過した値に付与する追跡子文字列を何らかの形式（プログラムへの特殊形式記述，GUI 上での操作）でリクエストする
2. 変換器は，デバッグ対象のプログラムの変換中に，変換後のプログラムに対してユーザのリクエストを満たすようなユーザ指定追跡子付与ポイントの設定も行う

5.4 追跡子を活用したアサーション機能

今回提案したデバッグ機構では，プログラムの実行終了後に，収集した値と追跡子の情報をクエリによって処理していたが，このような処理を実行時に行うことも可能である．収集した値と追跡子の情報を実行時に使用する処理としては，アサーションが考えられる．

追跡子を活用したアサーション機能としてまず考えられるのは，値が特定の追跡子を持つ/持たないことのアサーションである．これは，例えば式 exp と追跡子文字列 $marker$ を引数に取る関数 assertHasMkr，assertNhasMkr などと実装できる．関数 assertHasMkr は，第一引数の式 exp の評価結果の値が第二引数の追跡子文字列 $marker$ を持っていなかった場合，アサーション失敗として報告する．関数 assertNhasMkr はその逆のことを行う．

このようなアサーション機能の利用例を，ソースコード 5.2 に示す．この例では，配列の各要素のバリデーションを行う validate 関数の最後の処理で，assertHasMkr を使用している．これにより，validate 関数の処理にバリデーション漏れが無いことを確認できる．また，バリデーション漏れがあり，アサーションが失敗した場合には，アサーション失敗時の値を起点として，意図しない値のデバッグを行えるようにしておくと便利であると考えられる．

ソースコード 5.2　追跡子を活用したアサーション

```
function validate(arr) // 配列の要素のバリデーションを行う関数
  /* ここから何らかのバリデーション処理 */
  /* バリデーションした要素には追跡子文字列'validated'を付与するよう指定 */
  ...
  /* ここまで何らかのバリデーション処理 */
  arr.forEach(e => {
    assertHasMkr(e, 'validated'); // 各要素がバリデーションされたか確認
  });
}
```

第 6 章

関連研究

6.1 動的プログラムスライシング

動的プログラムスライシング [6] は，実行された命令と，その命令間の依存関係を表現した動的依存グラフ（DDG; dynamic dependence graph）から，指定した命令（間違った実行結果を出力した命令など）に依存関係のある部分グラフ（動的スライス）を取得する動的解析手法である．DDG では，命令間の依存関係として以下の 2 種類がある．

- データ依存関係：命令 X で変数 v を定義し，途中で再定義されずに命令 Y で変数 v を使用した場合，Y から X にデータ依存関係がある
- 制御依存関係：X が条件式で，X の評価結果によって命令 Y が実行されるかどうかが決まる場合，Y から X に制御依存関係がある

本研究では，追跡子を付与した値全体はグラフ構造を形成したが，これは DDG と類似したグラフなのではないかと考えている．特に，本手法における値の生成源の値を記憶する追跡子によって，DDG におけるデータ依存関係，制御依存関係に似た関係が表現されているのではないかと考えられる．

6.2 アルゴリズミックデバッギング

アルゴリズミックデバッギング [11] は，論理型・関数型言語で記述されたプログラムに適用するデバッグ技術として初めて発表されたが，PELAS [9] では，動的スライスを利用することによって，手続き型言語 Pascal で記述されたプログラムに対しての適用が実現されている．このアルゴリズムでは，間違った実行結果を出力した命令から動的スライスを遡ることによって，間違った実行結果を出力した原因となった命令を特定している．このアルゴリズムは以下の手順で動作する．

1. 間違った実行結果を出力した命令 X からデバッグを開始する．
2. 命令 X に依存関係のある命令の集合を S_X とし，調査対象の命令の集合 S に対して，$S = S_X$ とする．

3. S の各要素に対して，その命令の評価結果が正しいかどうかを確認する．
   (a) S の要素で評価結果が間違っている命令 Y があった場合，$S = S_Y$ として 3. を繰り返す．
   (b) S の要素で評価結果が間違っている命令がなかった場合，X をバグの原因と結論付ける．
   (c) S が空集合となった場合，バグの特定は失敗．

この過程は，本手法における意図しない値のデバッグの過程と類似している．実際，上記の過程を本手法では以下のように模倣できる．

1. 間違った実行結果を出力した値 v からデバッグを開始する．値 v の取得には，クエリ `findValue` などが利用できる．
2. 値 v の生成源の値をクエリ `findGen` によって求める，ここで求まった値の集合 V を調査対象とする．
3. V の各要素に対して，その値が正しいか（意図した値か）どうかを確認する．
   (a) V の要素で意図しない値 w があった場合, 値 w の生成源の値をクエリ `findGen` によって求め，求まった値の集合に対して 3. を繰り返す．
   (b) V の要素で意図しない値がなかった場合，v を評価結果とした式をバグの原因と結論付ける．
   (c) V が空集合となった場合，バグの特定は失敗．

6.3 プログラム実行時の情報を活用したデバッグツール

本ツールでは，プログラム実行時に収集された情報は，ユーザが実行したクエリによって適切に処理して提供することで，デバッグに有効活用されている．このようなプログラム実行時の情報を，ユーザの問い合わせに応じて適切に処理して提供するようなデバッグツールとして Whyline [8] がある．Whyline では，ユーザはデバッグ対象のプログラムの命令を指定し，なぜその命令が実行された/されなかったのかをデバッグシステムに問い合わせることができる．デバッグシステムは，問い合わせに対しての答えを動的スライスを利用して特定し，その情報を視覚的なグラフとしてユーザに提供する．

6.4 追跡子のアイデアを利用した研究

本研究では，デバッグへの活用のために追跡子のアイデアを使用してプログラム実行時の値の情報を取得したが，この用途以外で追跡子のアイデアを利用している研究もある．

権藤らの解析器 TBCppA [13] では，C 言語のプリプロセッサ処理前後の字句の対応関係を取得するために XML 風の文字列を追跡子として利用している．具体的には，マクロ定義部のマクロボティの場所と，マクロ呼出の場所に対して追跡子を付与し，プリプロセッサ処理後にそれぞれの追跡子がどのような組み合わせで付与されたかの情報から，どのマクロボティがどのマクロ呼出箇所に展開されたのかを調べる．このような付与された追跡子の組み合わせの情報は，本手法においてもプログラムの性質の検査において活用されている．

6.5 その他

5.4 節で示したような，元のプログラミング言語の構文・言語機能からはアクセスできない，プログラム実行中に存在している情報を活用してアサーションを行うための言語として DeAL[10] がある．DeAL は GC 実行によるヒープオブジェクトの走査中に行うアサーションを記述するための言語である．DeAL によるアサーションの評価は，GC 中に取得できる，ヒープの形状（それぞれのオブジェクトがどのように参照されているか）の情報を使用して実現している．

動的情報流解析 [7] では，プログラム実行時に，機密情報を含む値の情報が直接的・間接的に流出していないかどうかを解析する．その解析の際には値に機密度を表すラベルを付与するという技法を用いており，値に追跡用の追加データを取り付けるという技法は本手法以外においても用いられている．

第 7 章

おわりに

プログラム実行時の値がどのようなふるまいを行っているかを特定することは，プログラムの理解やデバッグの際に重要である．

本研究では，プログラム実行時の値の情報を取得し，それらの情報を適切に処理することによって，デバッグなどの解析に役立つ情報を得るための手法を示した．プログラム実行時の値の情報は，値に追跡子を付与するという考え方によって実現し，それらの情報を有効活用するためのクエリ言語を提案した．また，本手法を実装したデバッグツールを実際にバグがあるプログラムに対して適用する例を用いて，本手法の有用性を示した．本研究のデバッグツールでは，プログラムの字面上の場所からの値の取得や，値が実際に通過した経路をプログラム上に描画することができるため，実際のプログラムとそのプログラム実行時の値の対応関係を視覚的に理解することができる．また，意図しない値を発見したときは，その値が持つ追跡子を調べることによって，その値の生成源の値を知ることができる．これにより，値の流れを逆向きに遡ってバグの原因となった値を探ることができる．また，ユーザ指定の追跡子とクエリによって，プログラムの性質の検査を行うことができる．

本研究では，プログラム実行時に現れる個々の値をすべて区別しているため，プログラム実行時の値を収集したデータ量は巨大化しがちである．そのため，今後の課題として，収集する値の情報を適切に削減する手法を実現することが挙げられる．ただし，収集データ量の削減によって，実際のプログラム実行時の情報が部分的に失われてしまうため，削減後のデータからデバッグなどの解析を効果的に行えるかどうかは不明である．そのため，データ量削減の粒度と実行時情報の収集範囲の切り替えを行いながら，関心のあるプログラムの部分を絞り込んでいけるようなデバッグを行えるようにすることが望ましいと考えている．

参考文献

[1] Babel. https://babeljs.io/, 2022/1/22 現在.

[2] JavaScript. https://www.ecma-international.org/publications-and-standards/standards/ecma-262/, 2022/1/22 現在.

[3] Scheme. https://small.r7rs.org/attachment/r7rs.pdf, 2022/1/22 現在.

[4] クイックソートは難しい？ ： 文科省研修教材のコードには虫がいるよ. https://note.com/evjunior/n/n25c3e831d110, 2022/3/16 現在.

[5] 高等学校情報科「情報 I」教員研修用教材（本編）. https://www.mext.go.jp/content/20200722-mxt_jogai02-100013300_005.pdf, 2022/3/16 現在.

[6] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *SIGPLAN Not.*, Vol. 25, No. 6, pp. 246–256, jun 1990.

[7] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, p. 113–124, New York, NY, USA, 2009. Association for Computing Machinery.

[8] Amy J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, p. 301–310, New York, NY, USA, 2008. Association for Computing Machinery.

[9] B. Korel. PELAS—program error-locating assistant system. *IEEE Transactions on Software Engineering*, Vol. 14, No. 9, pp. 1253–1260, 1988.

[10] Christoph Reichenbach, Neil Immerman, Yannis Smaragdakis, Edward E. Aftandilian, and Samuel Z. Guyer. What can the GC compute efficiently? a language for heap assertions at GC time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, p. 256–269, New York, NY, USA, 2010. Association for Computing Machinery.

[11] Ehud Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, USA, 1983.

[12] Olin Grigsby Shivers. *Control-Flow Analysis of Higher-Order Languages of Taming Lambda*. PhD thesis, USA, 1991. UMI Order No. GAX91-26964.

[13] 権藤克彦, 川島勇人, 今泉貴史. TBCppA: 追跡子を用いた C 前処理系解析器. コンピュータ ソフトウェア, Vol. 25, No. 1, pp. 105–123, 2008.

謝辞

本研究を遂行するにあたっては，いろいろな方々にお世話になりました．

まず，主任指導教員の小宮常康先生には日頃から熱心なご指導，そしてご鞭撻を賜わりました．また，ご多忙中にもかかわらず論文の草稿を丁寧に読んで下さり，大変貴重なご助言をいただきました．ここに厚く御礼申し上げます．

また，副指導教員の久野靖先生には，論文を丁寧に読んで下さり，大変貴重なご助言をいただきました．ここに感謝申し上げます．

そして，本研究が行なえたことは，研究方針や方法論について議論をし，共に研究生活をおくってきた小宮研の学生諸氏おかげでもあります．最後に，これらの皆さんに感謝いたします．