情報処理学会論文誌　プログラミング　Vol.17　No.4　10–20 (Aug. 2024)

保守的なawaitのために同期的か非同期的かを実行時に切り替えるpotentially-async関数

川向 聡[1,a)]　山崎 徹郎[1,b)]　千葉 滋[1,c)]

受付日 2024年2月17日, 採録日 2024年5月14日

概要：この論文では保守的に，非同期的かもしれない関数呼び出しに await を書くことで await 忘れを防ぐという方針について考える．JavaScript の async/await 構文は非同期計算を記述するために便利であるが，しばしば await を書き忘れることが難しいバグの原因となる．しかし保守的な await では，同期的な式の await のために計算効率が大幅に低下してしまう．我々は不要な await によるオーバーヘッドを削減するため，potentially-async 関数を提案する．Potentially-async 関数では await 式は，オペランドが同期的か非同期的かを実行時に判断することで不要な await によるオーバーヘッドを削減できる．我々は実験によって不要な await が多いプログラムであれば実行時間を最大で 90%程度削減できること，必要な await が多いプログラムでは最大で 73%のオーバーヘッドが生じることを確認した．Potentially-async 関数はプログラム変換によって JavaScript プログラムに変換してから実行できるため，ブラウザ上でも実行することができる．

キーワード：async/await，プログラム変換，プリプロセス

Potentially-async Function that Switches between Synchronous and Asynchronous at Runtime for Conservative Awaiting

Satoshi Kawamukai[1,a)]　Tetsuro Yamazaki[1,b)]　Shigeru Chiba[1,c)]

Received: February 17, 2024, Accepted: May 14, 2024

Abstract: In this paper, we consider about an approach that conservatively writes awaits for potentially asynchronous function calls. Although async/await notation in JavaScript is useful to describe asynchronous computations, missing awaits often cause difficult bugs. However, conservatively awaiting significantly reduces execution speed due to awaiting for synchronous expressions. For reducing overheads due to unnecessary awaits, we propose *potentially-async function*. Potentially-async function can reduce the overhead due to unnecessary awaits because an await-expression changes its behavior depending on whether the operand is synchronous or asynchronous. Our experimental result shows that potentially-async function reduces execution time of programs with many unnecessary awaits by 90% at most, and it also causes another overhead that increases execution time of programs with many necessary awaits by 73% at most. We also provide a program transformation from potentially-async function declaration to async or non-async function declaration, thus one can execute potentially-async functions on their browsers.

Keywords: async/await, program transformation, preprocessing

[1] 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113–8654, Japan
[2] 情報処理学会プログラミング研究会
Information Processing Society of Japan Special Interest Group on Programming, Chiyoda, Tokyo 101–0062, Japan
[a)] skawamukai@csg.ci.i.u-tokyo.ac.jp
[b)] yamazaki@csg.ci.i.u-tokyo.ac.jp
[c)] chiba@acm.org

1. はじめに

多くの処理系がシングルスレッドモデルで動作する JavaScript では，スループット向上のために非同期処理のためのプリミティブが用意され，利用されている．ECMAScript 2017 で導入された async/await 構文によって非

同期処理を直感的に記述できるようになった [3].

この論文では await 忘れによって生じるバグを抑制するため，保守的に await が不要な式にも await を書くという方針について考える．Async/await 構文では，非同期的な式は `await` と修飾するかどうかによってまったく異なる意味となるが，`await` は書き忘れてもしばしばうまく動作してしまい，後にプログラムを書き換えたタイミングで表面化する，潜在的なバグとなる場合がある．`await` の要・不要を機械的に判定できれば `await` 忘れを防ぐことができるが，JavaScript のような動的な言語では完全な解析は困難であり，`await` 忘れを見逃す可能性はつねに残る．そこで我々は，`await` の数を増やすことで `await` 忘れを防ぐというプログラミングスタイルの可能性を検討することにした．

保守的な await の第 1 の課題は実行時間に大きなオーバーヘッドが生じる点である．極端に多くの `await` を書いてみる我々の実験では，ベンチマークの実行時間は 30 倍にも増大した．このオーバーヘッドは，不要な `Promise` オブジェクトの生成と，それにともなうコンテキストスイッチが原因だと予想される．

この論文では，不要な await による計算時間へのオーバーヘッドを削減するため，potentially-async 関数を提案する．Potentially-async 関数の中では，`await` 演算子は修飾された式が同期的か非同期的かをその返り値が `Promise` オブジェクトだったかどうかによって判定し，非同期的だった場合にのみ `Promise` オブジェクトを生成し，実行コンテキストをスイッチする．そのため，保守的に多くの await が書かれた関数を potentially-async 関数とすることでオーバーヘッドの削減が見込める．不要な await は同期的な式を修飾するため，`Promise` オブジェクトの生成と実行コンテキストのスイッチが起きなくなり，高速化が期待できる．保守的に await を書く状況として，プログラマが手動で挿入する場合や，ツールが機械的に await を挿入する場合などが考えられる．

Potentially-async 関数の特徴はコード変換によって通常の JavaScript プログラムに変換できる点にある．JavaScript 処理系の改造をともなわずに実行できるため，potentially-async 関数を含むプログラムを既存のブラウザ上で実行することも可能である．

以降のこの論文では，2 章で不要な await によって実行時性能が悪化する問題を示し，3 章で我々の提案である potentially-async 関数の概要と JavaScript プログラムへと変換する方法を示す．4 章では potentially-async 関数を変換することで得られたプログラムの性能を調べる実験を示し，5 章では関連研究を示す．6 章ではまとめと今後の課題を示す．

2. モチベーション

ECMAScript 2017 以降の JavaScript では，非同期処理を記述するために async/await 構文を利用することができる [3]．I/O や通信のような CPU への負荷は小さいがしかし時間のかかる処理を非同期的に行うことで，計算リソースのムダを削減しスループットを向上させることができる．JavaScript で非同期処理を書くためにコールバック関数や `Promise` などの API が使用されていたが，async/await 構文の導入によってより直感的に非同期処理を記述できるようになった．

たとえばプログラム 1 は async/await 構文を用いた非同期処理の例である．このプログラムで定義される `fetchData` は，引数に渡されたサーバーの URL に対して GET リクエストを送信し，サーバーから返却されたデータを構文解析し，解析結果を返す関数である．`fetchData` は async/await 構文を使用して非同期に振舞うように定義されているため，サーバーからの返事を待つ間に別の計算を進めることができる．`fetchData` のように，宣言が `async` と修飾された関数を async 関数と呼ぶ．async 関数の本体では非同期的な式を `await` と修飾することで，処理の完了を待ち，結果を得てから続きの計算を行わせることができる．たとえば 2 行目の `request(url)` は非同期にサーバーへリクエストを送信し，`await` によって応答を待つ．`await` で応答を待つ間はコンテキストスイッチし，別の非同期的な計算を進めることができる．

非同期的な式を `await` と修飾せずに書くこともできる．`await` と修飾していない非同期的な式は，非同期的であるため評価を始めてすぐに計算結果が得られるとは限らない．このような式では計算結果を表すふつうの値を返す代わりに，非同期的な計算が完了した後に結果が格納される `Promise` オブジェクトを返す．JavaScript では `Promise` オブジェクトの計算が完了したかを検査したり，計算結果を取り出したりする機能は提供されない．計算結果を取り出す代わりに，`.then()` メソッドによって計算の完了後にコールバックされる関数を登録することができる．`.then()` メソッドで登録したコールバック関数は計算結果を引数に呼び出されるため，結果を使用する計算を記述することができる．`.then()` メソッドはコールバック関数が登録された後の `Promise` オブジェクトを返却する．

プログラム 1　async/await を用いた非同期処理の記述

```
1 async function fetchData(url) {
2   const { data, res } = await request(url);
3   if (res.status !== 200) {
4     throw new Error("request failed");
5   }
6   return parseData(data);
7 }
```

プログラム 2　await 忘れによるバグの例

```
1 async function lookupKeys(keys) {
2     let values = [];
3     for (let k of keys) {
4         fetchValue(k).then((v) => {
5             values.push(v);
6         })
7     }
8     return keys.map((k, i) => [k, values[i]]);
9 }
```

async/await 構文は非同期処理を直感的に記述できるものの，どの式が非同期的なのかをプログラマが判断し，await と修飾する必要がある．非同期的な式を修飾し忘れた場合，様々なバグとなって現れる．たとえばプログラム 1 の 2 行目の await を書き忘れた場合，2 行目で request から返却される Promise オブジェクトが，そのまま data と res に分割代入される．Promise オブジェクトは data も res も持たないが，JavaScript ではこの分割代入は**エラーにならず**，data にも res にも undefined が格納される．3 行目の res.status に到達して初めて「undefined のプロパティを読むことはできません」とエラーが生じるが，このエラーメッセージから原因が await を書き忘れたことにあると類推するには慣れが必要である．

await の書き忘れは正しく計算できているように見えてしまうより根の深いバグの原因となることもある．たとえばプログラム 2 は await を書き忘れたプログラムの例である．lookupKeys は引数にキーの配列を受け取り，サーバーに記録されている値と組み合わせてキーと値のペアの配列を作成して返す関数である．3 行目の for 文でキーを順に取り出し，4 行目の fetchValue 関数で値を要求するリクエストをサーバーに送信する．さらに .then() メソッドによって，サーバーからの応答が返ってきた後に返却された値を値の配列 values に格納する処理を，fetchValue の続きとして登録している．そして 8 行目でキーの配列と値の配列を組み合わせ，ペアの配列を作成して返却する．

たとえばプログラム 2 は簡単なテストでは正しく動作しているように見えるが，しかし本番環境では正しく動作しない．4 行目の fetchValue やその後の .then() メソッドで登録される続きの計算は非同期的に実行されるが，lookupKeys 関数はこれらの非同期的な計算の完了を待たずに計算を進める．その結果，values 配列に要素が格納される前に 8 行目の return 文に到達してしまい，結果として値の部分がすべて undefined で埋められた配列が返却される．さらに厄介なことに，fetchValue がコンテキストスイッチをともなわない単純なモックだった場合，lookupKeys は期待通りに動作してしまう．これは，ECMAScript [*1] で async 関数の呼び出し時に，async 関数の本体を評価する Promise を execution context stack の上に追加するため，呼び出しの次の計算は async 関数の本体の評価と定められているためである．

本論文では await 忘れによるバグを抑制するため，保守的に await を挿入することを考える．JavaScript の async/await では await と修飾される式は必ずしも非同期的である必要はない．ECMAScript 2023 では，普通の同期的な式が await と修飾されていた場合はその式を評価した後，抽象操作 *Await* の先頭で *PromiseResolve* によって Promise オブジェクトに変換してから残りの処理を行うとされている．そのため，同期的な式を await と修飾しても計算結果は同じになる．

保守的に await を書く状況として，プログラマが手動で挿入する場合や，ツールが機械的に await を挿入する場合などが考えられる．もし非常に小さなオーバーヘッドで同期的な式を await できるなら，async/await に関するバグを発見したプログラマがとりあえずほとんどの式に await を書いてみる，といった使い方もできるようになるだろう．また，await 忘れによるバグを抑制するため，機械的に await を挿入するツール [2] を使用するという可能性も考えられる．このようなツールはある式が同期的か非同期的かを静的解析で決定できなかった場合，しばしば保守的に await を挿入する．非同期的かどうかが不明な式であっても小さな実行時オーバーヘッドで型エラーを回避できる方法を開発することで，await を自動的に挿入するツールの設計でも選択肢を増やすことができる．

しかし，過剰な await 演算子の挿入は余分な Promise オブジェクトの生成とコンテキストスイッチを誘発するため，実行速度にオーバーヘッドが生じる．4.3 節で示す実験では，修飾可能なすべての関数呼び出しを await と修飾した場合，11 倍から 30 倍ものオーバーヘッドが生じることが観察された．

await が不要だった場合の実行時間へのオーバーヘッドを抑えることができれば保守的な await の挿入の実現に 1 歩近づくが，このようなシステムを設計するにあたって注意するべき点が 2 つある．まず，JavaScript エンジンの改造を避け，プログラム変換だけで実現することが望ましい．JavaScript は広く使われているため，エンジンの改造が現実に使用できるようになるにはプロポーザルとして議論し，ブラウザベンダーに採用される必要があるため年月がかかる．その一方でプログラム変換だけで実現できたなら，開発者側の小規模な変更だけで採用することができる．Web 開発では JavaScript プログラムをデプロイする前に babel や webpack によるプログラム変換をかけるのが一般的である．プログラム変換器の形で実現できるなら，これらのツールチェインに加えるだけでアプリケーションに組み込むことが可能である．

第 2 に，我々が提案するような機能追加のためのプログラム変換はしばしばコードサイズを増大させるが，web アプ

[*1] 少なくとも ECMAScript 2023 では．

リケーションではプログラムサイズの増大は一定以下であることが望ましい．Web アプリケーションでは JavaScript プログラムのサイズが増大することは通信量が増大することを意味するため，プログラムサイズの増大によって web サイトの読み込みが遅くなる，サーバーの負荷が増大するといった問題が引き起こされる．

3. Potentially-async 関数

我々は同期的か非同期的かを実行時に判断し，同期的だった場合は `Promise` オブジェクトを生成しない potentially-async 関数を提案する．Potentially-async 関数は async 関数と同じように宣言するが，最初の行に `` `perhaps async` `` と書く．Potentially-async 関数の宣言では，async 関数と同じように非同期的な式を `await` と修飾することで非同期的な計算の完了を待ち，値を得てから計算を再開させることができる．しかし async 関数とは異なり，同期的な式を `await` と修飾した場合は `Promise` オブジェクトを生成せず，コンテキストスイッチもしない．特に関数内の `await` と修飾された式がすべて同期的だった場合は，`Promise` オブジェクトではないただの値を返すこともある．このように potentially-async 関数では async 関数と比較して `Promise` オブジェクトの生成箇所が減るため，不要な `await` による処理速度の低下を軽減できる．

プログラム 3 に potentially async function の例を示す．`fetchData` は potentially-async 関数であるため，1 行目で関数宣言を `async` と修飾しており，その本体の先頭である 2 行目に `` `perhaps async` `` と書いている．3 行目の `request` の呼び出しは非同期的であるため `await` と修飾している．7 行目の `parseResult` の呼び出しは同期的だが，これも保守的に `await` と修飾している．Async 関数であれば 7 行目の `await` で `Promise` オブジェクトが生成されるが，この `fetchData` は potentially-async 関数であるため，`Promise` オブジェクトを生成せずに `parseResult` の結果をそのまま返す．3 行目の `await` は必要な `await` であるため，`fetchData` 関数全体としては `Promise` オブジェクトを返却する．

3.1 プログラム変換の方針

Potentially-async 関数はプログラム変換によって通常の JavaScript プログラムに変換してから実行する．そのため，ツールチェインにプログラム変換器を挿入するだけで JavaScript エンジンの改造なしに使用することができる．この節では potentially-async 関数の宣言を同期的な関数の宣言，あるいは async 関数の宣言に変換するプログラム変換の手順を示す．この節で示すプログラム変換は try-catch 文には対応しておらず，変換することができない．

この変換では，potentially-async 関数内のそれぞれの `await` について，修飾される式が `Promise` オブジェクトを返すかどうかを実行時に調べ，もし `Promise` オブジェクトを返すなら従来の async/await とほぼ同性能のスローパスに，`Promise` 以外のオブジェクトを返すならより高速なファストパスへと分岐させる．プログラム 3 を変換した結果得られる JavaScript プログラムをプログラム 4 に示す．22 行目の `v1` の宣言では `await` と修飾された 1 つ目の式である `request(url)` の結果を格納している．`v1` はその後，23 行目で `instanceof` 演算子によって `Promise` オブジェクトだったかどうかを検査し，`Promise` オブジェクトだった場合は 24 行目のスローパスに，`Promise` 以外のオブジェクトだった場合は 26 行目のファストパスへと分岐する．

ファストパスではそれが同期的な式だったかのように計算を続ければいいため，それ以上追加のバリアを挿入せず，単に続きの計算を行うプログラムを生成してファストパスとする．一方，スローパスでは async/await 構文を使うことができないため，`Promise.then()` メソッドを使用して `await` の動作をエミュレートする．スローパスで `await`

プログラム 3　Potentially async 関数

```
1 async function fetchData(url) {
2     `perhaps async`
3     const { data, res } = await request(url);
4     if (res.status !== 200) {
5         throw new Error("request failed");
6     }
7     return await parseData(data);
8 }
```

プログラム 4　変換後の JavaScript プログラム

```
1  function fetchData(url) {
2      const k1 = (v2) => {
3          const { data, res } = v2;
4          const k2 = (v3) => {
5              const k3 = (v5) => {
6                  return v5;
7              }
8              const v4 = parseData(data);
9              if (v4 instanceof Promise) {
10                 return v4.then(k3);
11             } else {
12                 return k3(v4);
13             }
14         }
15         if (res.status !== 200) {
16             throw new Error("request failed");
17             return k2();
18         } else {
19             return k2();
20         }
21     }
22     const v1 = request(url);
23     if (v1 instanceof Promise) {
24         return v1.then(k1);
25     } else {
26         return v1;
27     }
28 }
```

を使用するには関数宣言を async と修飾する必要があるが，async と修飾するとファストパスでも Promise オブジェクトを生成してしまう．Promise.then() メソッドに渡すために await 以降の継続を関数化するため，プログラム 4 では多くの文がプログラム 3 とは上下が逆の順番で現れる．

For 文や while 文などの繰り返し文の内側で await が使用されている場合は変換を中断し，最初の行の `perhaps async` を取り除いただけの async 関数へと変換する．これは，繰り返し文の途中から始まる継続と等価な関数を無理やり生み出すとファストパスとスローパス両方の実行速度が低下してしまうためである．スローパスでは Promise.then() メソッドによって await の動作をエミュレートするため，await と修飾された式の継続を関数として切り出す必要がある．しかし，繰り返し文の途中で await が使用されている場合，その継続は繰り返し文の途中から始まるため，プログラムの一部を抜き出したような単純な式では表現できない．繰り返し文を等価な再帰関数に変換してから継続を関数化することで繰り返し文の途中から始まる継続と等価な関数を生み出すことも可能だが，このような再起呼び出しは for 文や while 文より遅いため，ファストパスとスローパス，どちらも実行速度が低下してしまう．

たとえば，プログラム 5 の (a) に繰り返し文の中に await を含む関数を，(b) に for 文を再帰関数に変換することで得られた await の継続を示す．途中から始まる for 文の 1 周目だけをアンロールすることでも await の継続が得られると思うかもしれないが，この方法で得られた継続は await を含む．この await を再帰的に変換した場合，その結果は (b) と同じ関数となる．

プログラム 5　繰り返し文中の await の継続

```
1  // (a) 繰り返し文中に await を含む関数
2  async function await_in_loop(n) {
3      let local_var = 0;
4      for (let i = 0; i < n; i++) {
5          prologue();
6          local_var += await task(i);
7          epilogue();
8      }
9      return local_var;
10 }
11
12 // (b) for 文を再帰関数に変換して得た継続
13 function cont(n, local_var, i, v) {
14     local_var += v;
15     epilogue();
16     i++;
17     if (i < n) {
18         prologue();
19         let v_next = task(i)
20         return cont(n, local_var, i, v_next);
21     } else {
22         return local_var;
23     }
24 }
```

プログラムサイズの指数的な増大を防ぐため，変換の中で複製されるプログラム片を関数化する．複製されるプログラム片には await 式の継続と if 文より後のプログラム片の 2 種類がある．たとえば await 式の継続はファストパスとスローパスの両方に複製される．そのため，1 つの potentially-async 関数が複数の await を含む場合，変換後のプログラムのサイズは指数的に増大する．このようなプログラム片を関数化し，関数呼び出しで置き換えることでプログラムサイズの増大を変換前のサイズに対して線型程度に抑えることができる．

この節で示すプログラム変換は，直感的には await ごとに継続と等価な関数を構成し，それぞれの方法で継続を呼び出すファストパスとスローパスを生成する．そのため，変換前のプログラムと変換後のプログラムでコンテキストスイッチの有無を除いて同じように振舞うことが期待される．

3.2　変換の詳細

3.2.1　繰り返し文の中で await が使用されているかを検査

まず，for 文や while 文などの繰り返し文の中で await が使用されているかどうかを検査する．もし使用されているなら関数の 1 行目にある `perhaps async` を取り除いて変換を終了する．この検査は単に potentially-async 関数に対応する構文木を上から走査し，for-of 文や while 文を表す頂点の子孫に await 式を表す頂点が存在するかどうかを調べればよい．

3.2.2　条件分岐の継続を関数化・分配

次に if 文などの条件分岐の継続を無引数の関数に関数化し，継続の呼び出しを then 節と else 節に分配する．この変換のイメージ図をプログラム 6 に示す．斜体の e や s は変換前のプログラムの一部を表すメタ変数である．e は式を表し，s は文を表す．上線付きの $\overline{s}$ は次の行の ... と合わせて 0 個以上の文を表す．*cont* は他の識別子と衝突しないように生成したユニークな識別子を表す．たとえば 1 つの potentially-async 関数に 2 つの if 文が含まれていた場合では，1 つ目の if 文の変換から生まれた *cont* と 2 つ目の if 文の変換から生まれた *cont* は互いに異なる識別子となる．

この変換では if 文を 1 つずつ順番に変換する．If 文がネストしていた場合は，より外側の if 文から順に変換する．If 文がネストせずに並んでいた場合はより前の if 文を先に変換し，その後生成される *cont* の宣言を再帰的に変換する．

この変換では継続を表す関数 *cont* の宣言とその呼び出しを追加する．JavaScript の if 文は値を返さないので，*cont* は無引数関数で十分である．*cont* の中身は if 文の後に続く文の並び $\overline{s_3}$; ... である．*cont* の定義の中に明示的には

プログラム 6　if 文の継続の関数化と分配

```
// 変換前のプログラム片
if (e) {
  s1;
  ...
} else {
  s2;
  ...
}
s3;
...

// 変換後のプログラム片
function cont() {
  s3;
  ...
}
if (e) {
  s1;
  ...
  return cont();
} else {
  s2;
  ...
  return cont();
}
```

プログラム 7　await の継続の関数化

```
// 変換前のプログラム片
let sum = await e1 + await e2;
s;
...

// 変換後のプログラム片
function cont(v) {
  let sum = v + await e2;
  s;
  ...
}
return cont(await e1);
```

return 文を書いていないが，変換の対象が値を返す関数なら $\overline{s_3}$; ... の中に return 文が含まれているため，変換後の関数も値を返す．

Else 節が存在しない場合は，空の else 節を作成して継続の呼び出しを挿入する．Else-if 節については，else 節の中に if 文がネストしていると考えて変換を行う．

Switch 文の場合も同様に変換を行う．Switch 文の場合に継続呼び出しを挿入するのは `break` の直前，存在するなら `default:` の直前，switch 文の末尾の 3 箇所である．

3.2.3　Await 式の継続を関数化

次に await 式の継続を関数化する．この変換のイメージ図をプログラム 7 に示す．ここでは一例として，await 式どうしの足し算の変換を示す．このプログラムでも e と s は変換前のプログラムの一部を表すメタ変数であり，*cont* と v はユニークな識別子である．

この変換でも await 式を 1 つずつ順番に変換する．Await 式が複数存在する場合は，計算順序に従って先に計算される await 式を先に変換し，その後生成される *cont* の宣言を再帰的に変換する．プログラム 8 は足し算の左辺の await 式を変換する場合を示している．そのため *cont* の中に右辺の await 式が継続が関数化されないまま残っているが，これは後で再帰的に変換される．

プログラム 8　`await` 式の変換

```
// 変換前のプログラム片
return cont(await e);

// 変換後のプログラム片
let v = e;
if (v instanceof Promise) {
  return v.then(cont);
} else {
  return cont(v);
}
```

この変換でも継続を表す *cont* 関数とその呼び出しを追加する．3.2.2 項の変換とは異なり値を返す式の変換であるため，*cont* は引数 v を受け取る．*cont* の中身は変換中の await 式を含む文の変換中の式を v で置き換えたものと，変換中の await 式を含む文より後の文の並び $\overline{s}$; ... からなる．

この変換では変換中の await 式を含む文より後の文を表す $\overline{s}$; ... はブロックの終わりまでを捕捉すれば十分であり，ブロックの外に続きが存在する可能性を考える必要はない．これは，3.2.1 項で示した検査と 3.2.2 項で示した変換によって，ここで変換の対象となっている `await` より後は，関数の終わりまで，どんな文も存在しないようにプログラムが変形されているためである．

3.2.4　Await 式を分岐に変換

最後に，await 式をファストパスとスローパスの分岐に変換する．この変換のイメージ図をプログラム 8 に示す．このプログラムでは e と *cont* は変換前のプログラムの一部を表すメタ変数であり，v はユニークな識別子である．

3.2.3 項の変換によってすべての await 式は `return cont(await e);` の形に変形されている．そのため，この形の文だけを変換すればよい．ファストパスでは e を評価した結果を引数に継続 *cont* を呼び出す．スローパスでもすでに継続は関数化してあるため，`e.then()` に継続 *cont* を渡すだけでいい．最後に最初の行の `` `perhaps async` `` と関数宣言を修飾していた `async` の 2 つを取り除いて変換は完了する．

4.　実験

我々は，potentially-async 関数を 3.2 節で示したコード変換によって変換して得られたプログラムの性能を調査するため，ベンチマークの実行速度を計測する実験を行った．また，関数化によるコードサイズの削減とファストパスへのオーバーヘッドを調査する実験も行った．この章では，

表 1　作成したベンチマークの一覧
Table 1　List of benchmark programs.

	クラス宣言の数	関数宣言 合計	関数宣言 ループ	関数呼び出し 合計	関数呼び出し 静的
ループの少ないマイクロベンチマーク					
Towers	2	6	1/6	9	0/9
List	2	5	0/5	12	0/12
ループの多いマイクロベンチマーク					
Bounce	2	3	1/3	13	4/13
Storage	1	2	1/2	3	0/3
マクロベンチマーク					
DeltaBlue	13	53	8/53	141	3/141
Richards	10	35	1/35	71	8/71
Json	10	62	6/62	119	8/119

これらの実験とその結果を示す．

4.1　ベンチマークプログラム

性能評価のため，我々は余分な await を含むベンチマークスイートを用意した．用意したベンチマークプログラムの一覧を**表 1** に示す．関数定義の合計の列にはベンチマークプログラムが含む関数宣言の数を，ループの列には関数宣言のうち，繰り返し文中に関数呼び出しを含むため，3.2.1 項の手順で変換が中止される関数の数を示す．関数呼び出しの合計の列にはベンチマークプログラムが含む関数呼び出しの数を，静的の列には関数呼び出しのうち，コンストラクタの中や静的変数の初期化など，非同期的には呼び出すことができない関数呼び出しの数を示す．どのベンチマークプログラムも Marr らの “Are We Fast Yet?” ベンチマーク [5] に含まれるベンチマークプログラムを元に，改造することで作成した．“Are We Fast Yet?” に含まれるベンチマークはどれも async 関数を含まない，同期的な JavaScript プログラムである．我々はこれらのベンチマークプログラムのすべての関数を async 関数に変更し，関数内のすべての関数呼び出しを `await` と修飾することで余分な `await` を含むベンチマークプログラムを作成した．

我々が作成したベンチマークプログラムではそれぞれの `await` が必要か不要かを切り替えることができるように，`await` と修飾する式には `wrap` 関数の呼び出しも挿入する．`await` を挿入する書き換えと `wrap` 関数の定義のイメージ図をプログラム 9 に示す．`wrap` 関数が引数のオブジェクトを `Promise.resolve` に渡すことで，すべての `await` と修飾される式が `Promise` オブジェクトを返すようになり，すべての await が必要なものになる．

コンストラクタは async 関数にできないため，コンストラクタから呼び出される関数は async 関数版と同期的版の 2 種類を用意した．コンストラクタから呼ぶ場合を同期的版を呼び，それ以外の場所から呼ぶ場合は async 関数版を呼ぶというように，文脈によってこれらを使い分ける．グローバル変数の初期化のために呼び出される関数についても同様である．

プログラム 9　`await` の挿入

```
1  // 書き換え前のプログラム片
2  let z = func(x, y);
3
4  // 書き換え後のプログラム片
5  let z = await wrap(func(x, y));
6
7  // wrap の宣言
8  function wrap(obj) {
9      // どちらかの行をコメントアウトする
10     // return Promise.resolve(obj);
11     return obj;
12 }
```

4.2　実験環境

我々は実験のために，3 章で示した potentially-async 関数の宣言を JavaScript プログラムに変換するコード変換器を実装した．構文解析には babel-parser を用いた．

ウォームアップの影響を取り除くため，ベンチマークプログラムの実行時間は 60 回実行し最初の 10 回を取り除いた 50 回の平均を計算した．分散は最大でも 2%程度と非常に小さかったため，実験結果のグラフにエラーバーは表示していない．

計測はどれも Intel(R) Xeon(R) W-2123 CPU@3.60 GHz と 64 GiB のメモリを搭載したマシン上で行った．OS は Ubuntu 22.04 である．JavaScript 処理系は Node.js を使用した．Node.js のバージョンは 18.7.0 である．

4.3　ファストパスの計算速度

我々はファストパスの計算速度を評価するため，`await` が不要な設定の我々のベンチマークを次の 3 種類の方法で書き換えてから実行し，計算時間を比較した．

同期的関数版　`async` と `await` をすべて取り除いてから実行

async 関数版　ベンチマークプログラムを書き換えずに実行

potentially-async 関数版　async 関数を potentially-async 関数だと思って 3.2 節で示した手順で変換してから実行

`await` が不要な設定の我々のベンチマークは，async 関数は含むがそれ以外の `Promise` を生成する計算を含まないため，単に `async` と `await` を取り除いただけの同期的な関数でも正しく計算できる．この `async` と `await` を取り除いた関数は，オリジナルの “Are We Fast Yet” のベンチマークと比較して `wrap` の呼び出しだけが追加されている．

図 1 に各ベンチマークの実行時間を示す．ベンチマークごとに，左が同期的関数版，中央が async 関数版，右が提

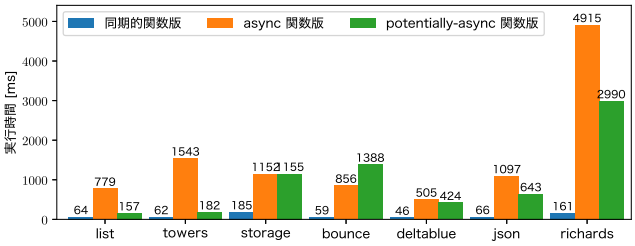


図 1　`await` が不要だった場合の実行時間
Fig. 1　Execution time when awaits are unnecessary.

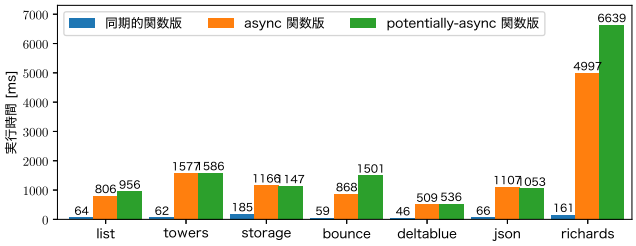


図 2　`await` が必要だった場合の実行時間
Fig. 2　Execution time when awaits are necessary.

案手法である potentially-async 関数版である．縦軸がベンチマークプログラムの実行時間を表しており，バーが短い方が短時間でベンチマークの実行を完了している．

`await` が不要な設定では，ベンチマークによって実行時間が同期的関数に近づく場合と async 関数に近づく場合とが観察された．list や towers のような繰り返し文の少ないベンチマークでは potentially-async 関数によって実行時間が最大で −90%程度まで削減できるなど，大きなオーバーヘッドの削減が観察された．ファストパスによる高速化の影響が強く現れたものと思われる．Potentially-async 関数版のこれらのベンチマークの実行時間は，同期的関数版と比較しても 1.5 倍や 2 倍と，無視できないオーバーヘッドは生じているものの比較できる程度の実行時間だった．

一方で storage や bounce のような繰り返し文の多いベンチマークでは，potentially-async 関数版の実行時間は async 関数版と同等か，あるいは +16%程度であった．繰り返し文の中に await を含む関数が多いため，変換された後のプログラムの大部分が async 関数版と同じだったものと思われる．

マクロベンチマークである deltablue，json，richards はそれぞれで傾向が異なるが，大まかには potentially-async 関数版の実行時間は同期的関数版と async 関数版の間くらいであった．マイクロベンチマークの結果からファストパスによる高速化の大小には関数との相性が見られるため，様々なタイプの関数が強調して動作するマクロベンチマークでは中間的な結果が得られたものと考えられる．

4.4　スローパスの計算速度

我々はスローパスの計算速度を評価するため，`await` が必要な設定の我々のベンチマークの async 関数版と potentially-async 関数版を実行し，計算時間を比較した．**図 2** にそれぞれの実行時間を示す．図 1 と同様に，左が同期的関数版，中央が async 関数版，右が提案手法である potentially-async 関数版，縦軸が実行時間である．await が必要なプログラムを同期的に実行することはできないため，左の同期的関数版の実行時間は参考値として図 1 と同じ値をプロットしている．

`await` が必要な設定では，potentially-async 関数版の実行時間は多くのベンチマークで async 関数版と比較して −5%から +19%と，似たような傾向を示した．しかし bounce と richards では，+73%と +33%と大きなオーバーヘッドが観察された．

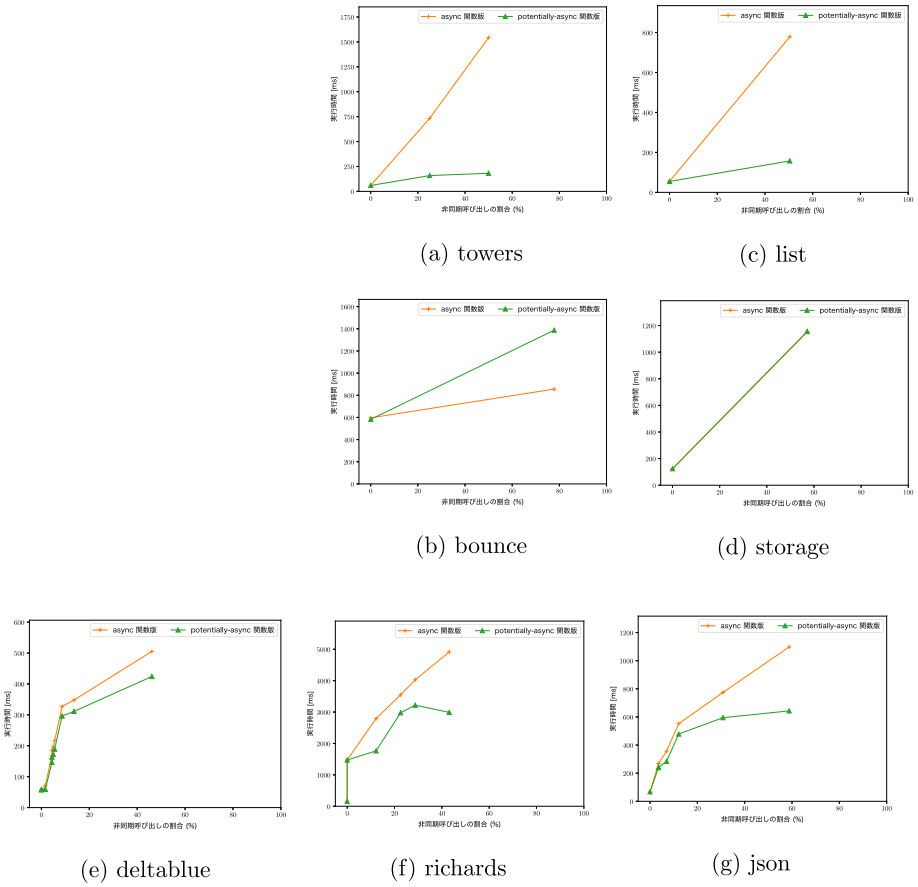
4.5　`await` の数とファストパスの実行時間の関係

さらに我々は，余分な `await` の数を変化させることがファストパスの実行時間に与える影響についても実験を行った．まず，4.1 節で示したベンチマークプログラムとは別に，関数ごとに同期関数か async 関数に振り分けることで `await` の数を変化させたベンチマークプログラムを作成した．同期的な関数は非同期的な関数を呼び出してもその結果を利用できないため，プログラムのエントリーポイントからコールグラフに沿って，根に近い関数を async 関数に，葉に近い関数を同期的な関数に振り分ける．Async 関数への変換を抑制する度合いを表すハイパーパラメータを L とする．L を変化させることで 1 つのベンチマークプログラムから，同じ計算をする `await` の数の異なる複数のベンチマークプログラムを生成した．葉までの距離が L 以下の関数は async 関数に変換せず，同期的関数のままにする．関数呼び出しが循環していた場合は，強連結成分を縮約したグラフの上で葉までの距離を計算する．

図 3 の (a) から (g) に各ベンチマークの実行時間を示す．縦軸が実行にかかった時間を，横軸は実行時に呼び出された関数のうち，async 関数の割合をそれぞれ表す．どのベンチマークでも async 関数の呼び出しが増えるにつれてオーバーヘッドが大きくなることが見て取れる．さらに，どのベンチマークでもベンチマークごとに，`await` の数を変化させても potentially-async 関数と async 関数の実行時間の関係は大きくは変化していない．Async 関数と同期的な関数が混ざった環境でも potentially-async 関数によってオーバーヘッドを削減することができた．

4.6　関数化の影響

3.2 節で示したプログラム変換では，プログラムサイズの爆発を防ぐために重複するコードの関数化を行った．重複するコードを 1 つの関数に共通化することでプログラムサイズを削減できるが，しかしファストパスに関数呼び出しのオーバーヘッドが生じる．この節では，関数化によっ



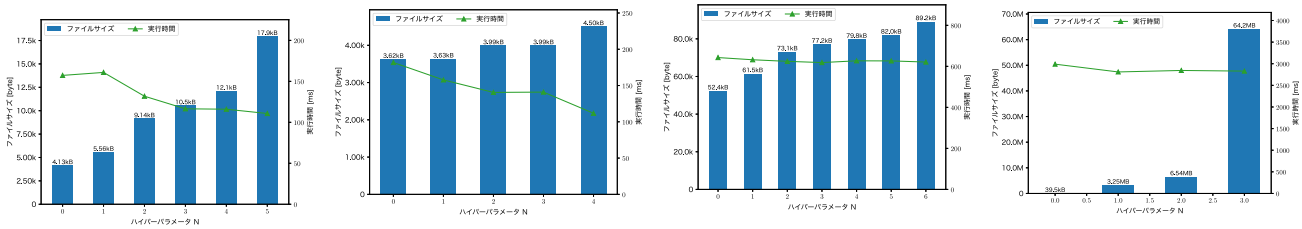
(a) towers　(c) list

(b) bounce　(d) storage

(e) deltablue　(f) richards　(g) json

図 3　await の数とファストパスの実行時間

Fig. 3　Relation between the number of awaits and execution time of fast path.



(a) list　(b) towers　(c) json　(d) richards

図 4　関数化による影響

Fig. 4　Impact to program size and execution time by functionalization.

てどれだけコードサイズを削減できたか，また関数化のためにファストパスにどれだけオーバーヘッドが生じたのかを調べる実験を示す．

我々は以下の方法で，potentially-async 関数を含むプログラムから関数化の度合いの異なる複数の JavaScript プログラムを生成した．関数化の度合いを表すハイパーパラメータを N とする．直感的には，このハイパーパラメータは 1 つの関数が継続を関数化せずに含むことができる最大の `await` の数を表す．3.2 節で示したプログラム変換に従って potentially-async 関数の宣言を potentially-async でない関数の宣言に変換する中で，3.2.3 項で示した `await` の継続の関数化で継続を関数化する `await` を選択することで関数化の度合いを調整する．

ある関数が A 個の `await` を含む場合，次のように継続を関数化させる `await` を選択する．まず，A が N 以下である場合は，それ以上継続を関数化せず，続くファストパスとスローパスの挿入では，継続に相当する文の並びをファストパスとスローパスの両方に複製する．もし A が N より大きかった場合は，ちょうど真ん中の `await` を選択し，継続を関数化する．`await` の数が偶数だった場合は，後ろ側の `await` を選択する．そして関数化された継続と選択した `await` より前のプログラム，それぞれを再帰的に変換する．

N を変化させながら以上の手順を繰り返すことで，関数

化の度合いの異なる複数のプログラムを作成した．**図 4** にこれらのプログラムのサイズと実行時間を示す．1 つの関数に含まれる `await` の数に対してプログラムのサイズは指数的に巨大化することが予測されたが，我々が実験した中では，list と richards はプログラムサイズが大きく変化したが，towers と json はプログラムサイズは大きくは変化しなかった．また，実行時間については関数呼び出しのオーバーヘッドが減るため高速化が期待されたが，list と towers では 20%程度削減できているものの json と richards ではほとんど変化しないという結果だった．

5. 関連研究

我々の提案する potentially-async 関数は保守的に await を書いた場合に生じるオーバーヘッドをコード変換だけで削減する．並列計算のための高速な同期プリミティブを提案する研究はいろいろある [4], [7] が，potentially-async 関数は同期の必要性を自動的に判定し，同期の機会を減らすことで高速化を狙う点が異なる．逆に potentially-async 関数を JavaScript 以外の環境に応用するには，同期の必要性を判定できるように意味論を制限する必要があるだろう．

JavaScript における非同期処理に由来するバグを検出する手法も提案されている [1], [6], [8], [9], [10], [11]．彼らは検出したバグを提示することで修正を促すのに対して，potentially-async 関数は await 忘れという特定の原因に対して，バグが生じにくいプログラミングスタイルを妨げる障害を取り除こうとしている点が異なる．静的検査によって await 忘れを検出できれば potentially-async 関数の有用性は損なわれるが，JavaScript の解析は困難であるため，検出できない await 忘れは存在する．静的検査と potentially-async 関数を組み合わせることで，可能性の低い await 忘れについては potentially-async 関数に置き換えて await を自動挿入することで，await 忘れによるバグを抑制するより優れたシステムを設計できる可能性がある．

6. まとめと今後の課題

我々は同期的な式にも保守的に await を書いた場合のオーバーヘッドを削減するため，`await` と修飾された式が非同期的かどうかを実行時に判断して `Promise` オブジェクトの生成とコンテキストスイッチの有無を切り替える potentially-async 関数を提案した．保守的に多くの await が書かれた関数を potentially-async 関数とすることで過剰な await によるオーバーヘッドを削減できることが期待される．

Potentially-async 関数のファストパスでの性能を調べる実験の結果，繰り返し文の中に `await` を含むかどうかによって得手不得手が大きく分かれるという結果が得られた．ファストパスの性能を調べる実験では，`await` を含む繰り返し文の少ないマイクロベンチマークでは −85%と −91%と，async 関数と比較してベンチマークの実行時間を大幅に削減することができた．一方で `await` を含む繰り返し文の多いマイクロベンチマークの実行時間では +0%と +16%と，オーバーヘッドが観察された．どちらの傾向を持つ関数も含むマクロベンチマークでは −15%から −48%程度，実行時間の短縮が見られた．総合的には高速化の影響が勝っているようである．

またスローパスの性能を調べる実験では，最悪の場合では実行時間に +73%と大きなオーバーヘッドが生じることが観察された．正確に必要な式だけを `await` と修飾している async 関数を potentially-async 関数に置き換えるとオーバーヘッドが生じることが予想される．

これらの実験は不要な `await` が極端に多いベンチマークで計測しているため，現実的なアプリケーションではより高速化の影響は小さくなるものと予想される．`await` の数を変えて実行時間を比較する実験では，`await` の数を減らすに従って async 関数と potentially-async 関数で性能が近づいていくという結果が観察された．

さらに関数化の度合いを変化させる実験では，関数化しなければプログラムサイズが増大する場合があることと関数化の度合いを下げると計算性能が向上する場合があるという結果が得られた．継続を共通化しないことにとるソースコードの増大は指数的であり，特に影響が大きかった richards は，共通化なしには Node.js では実行できないという結果が得られた．

今後の課題として，プログラム変換の健全性・完全性の検証や `await` を自動挿入するツールとの統合があげられる．3.1 節，および 3.2 節で示したプログラム変換は，我々が実験に使用したベンチマークでは動作しているものの，どんな JavaScript プログラムを変換してもそのふるまいが変化しないのかについては検証できていない．変換の前後でコンテキストスイッチの有無を除いて関数のふるまいが変わらないという健全性，およびどんなプログラムでも変換が成功するという完全性の検証が望まれる．

また，我々は余分な await による実行時間のオーバーヘッドを削減する手法を提案したが，これだけで await 忘れによるバグを抑制するためには，プログラマが保守的に多くの `await` を書く必要があり，手間である．Await 忘れの可能性を検出するような静的解析手法は提案されているため，解析の結果 await が忘れられている可能性はあるが見込みは低いと判定された式を potentially-async 関数に置き換えることで，プログラマへの負担は小さく，かつ実行時間への影響も抑えた await 忘れの予防手法が開発できるのではないかと期待される．

謝辞　本研究は JSPS 科研費 JP20H00578，JP24H00688 の助成を受けたものです．

参考文献

[1] Alimadadi, S., Zhong, D., Madsen, M. and Tip, F.: Finding broken promises in asynchronous JavaScript programs, *Proc. ACM Program. Lang.*, Vol.2, No.OOPSLA (online), DOI: 10.1145/3276532 (2018).

[2] Beillahi, S.M., Bouajjani, A., Enea, C. and Lahiri, S.: Automated Synthesis of Asynchronizations, *Static Analysis*, Singh, G. and Urban, C. (Eds.), Springer Nature Switzerland, pp.135–159 (2022).

[3] Ecma International: ECMAScript(R) 2017 Language Specification (2017), available from ⟨https://262.ecma-international.org/8.0/⟩.

[4] Imam, S. and Sarkar, V.: Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns, *Proc. 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, Berlin, Heidelberg, Springer-Verlag, pp.618–643 (online), DOI: 10.1007/978-3-662-44202-9_25 (2014).

[5] Marr, S., Daloze, B. and Mössenböck, H.: Cross-language compiler benchmarking: Are we fast yet?, *Proc. 12th Symposium on Dynamic Languages*, *DLS 2016*, pp.120–131, Association for Computing Machinery (online), DOI: 10.1145/2989225.2989232 (2016).

[6] Rau, O., Voss, C. and Sarkar, V.: Linear Promises: Towards Safer Concurrent Programming, *35th European Conference on Object-Oriented Programming* (*ECOOP 2021*), Møller, A. and Sridharan, M. (Eds.), Leibniz International Proceedings in Informatics (LIPIcs), Vol.194, pp.13:1–13:27, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (online), DOI: 10.4230/LIPIcs.ECOOP.2021.13 (2021).

[7] Shirako, J., Peixotto, D.M., Sarkar, V. and Scherer, W.N.: Phasers: A unified deadlock-free construct for collective and point-to-point synchronization, *Proc. 22nd Annual International Conference on Supercomputing*, *ICS '08*, pp.277–288, Association for Computing Machinery (online), DOI: 10.1145/1375527.1375568 (2008).

[8] Sotiropoulos, T. and Livshits, B.: Static Analysis for Asynchronous JavaScript Programs, *33rd European Conference on Object-Oriented Programming*, *ECOOP 2019*, Donaldson, A.F. (Ed.), Leibniz International Proceedings in Informatics (LIPIcs), Vol.134, pp.8:1–8:29, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (online), DOI: 10.4230/LIPIcs.ECOOP.2019.8 (2019).

[9] Sun, H., Bonetta, D., Schiavio, F. and Binder, W.: Reasoning about the Node.js event loop using async graphs, *Proc. 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, *CGO 2019*, pp.61–72, IEEE Press (2019).

[10] Tominaga, E., Arahori, Y. and Gondow, K.: AwaitViz: A visualizer of JavaScript's async/await execution order, *Proc. 34th ACM/SIGAPP Symposium on Applied Computing*, *SAC '19*, pp.2515–2524, Association for Computing Machinery (online), DOI: 10.1145/3297280.3297528 (2019).

[11] Turcotte, A., Shah, M.D., Aldrich, M.W. and Tip, F.: DrAsync: Identifying and visualizing anti-patterns in asynchronous JavaScript, *Proc. 44th International Conference on Software Engineering*, *ICSE '22*, pp.774–785, Association for Computing Machinery (online), DOI: 10.1145/3510003.3510097 (2022).

川向 聡

2024 年東京大学大学院情報理工学系研究科修士課程修了．修士（情報理工学）．同校では JavaScript の非同期プリミティブを改良する研究に従事．現在，都内の IT 企業に勤務．

山崎 徹郎　（正会員）

2021 年東京大学大学院情報理工学系研究科博士課程修了，博士（情報理工学）．現在，同研究科助教．メタプログラミングの応用，FFI 環境下でのガベージコレクションの研究に従事．

千葉 滋　（正会員）

1991 年東京大学理学部情報科学科卒業，1996 年同大学理学系研究科情報科学専攻より博士（理学）．2012 年東京大学情報理工学系研究科創造情報学専攻教授．プログラミング言語，システムソフトウェアの研究に従事．