不要な場合でもオーバヘッドの小さい await の JavaScript 言語における実現に向けて

川向 聡　山崎 徹郎　千葉 滋

本論文では，同期関数の呼び出しに付与した場合でもオーバヘッドの小さい新たな await の JavaScript における実現方法を提案する．JavaScript の await は本来，非同期関数の終了を待つための演算子だが，どの関数が非同期関数であるかは自明ではない．await の書き忘れによるバグを避けるため，すべての関数呼び出しに await を付与することが有効である．しかし，同期関数の呼び出しに本来不要である標準の await を付与すると，実行速度が大きく低下する．我々が提案する新たな await では，Promise オブジェクトを処理するスローパスと，高速なファストパスを設ける．このファストパスは新たな await が付与された式が Promise オブジェクトを返さなかった場合に実行され，オーバヘッドを抑える．

1　はじめに

JavaScript では，非同期処理を扱うために Promise というオブジェクトが導入されている．Promise オブジェクトを明示的に操作する代わりに async/await 構文を用いると，非同期処理をより簡単に記述することができる．Promise オブジェクトは非同期処理が完了したかどうかの状態を持つオブジェクトであり，await は Promise オブジェクトの完了を待つ演算子である．

標準の await は，Promise オブジェクトを返さない式に付与したとき，オーバヘッドが大きいという問題点がある．例えば async/await 構文を用いて非同期処理を記述するとき，Promise オブジェクトを返す関数の呼び出しに対して await を書き忘れることによるバグがしばしば生じる．この対策として，Promise オブジェクトを返すものと返さないものを区別せず，すべての関数呼び出しに保守的に await を付与することが有効と考えられるが，標準の await を利用するとプログラムの実行速度が大きく低下してしまう．

本論文では，この問題を解決するために Promise オブジェクトを返さない式に付与してもオーバヘッドの小さい await?という演算子を新たに設け，await?を JavaScript 上で実現するためのコード変換について提案する．提案する await?は，コード変換によってスローパスとファストパスという 2 つのパスをプログラムに挿入し，Promise オブジェクトを返さない式に付与されていた場合は実行時間の短いファストパスへ分岐することでオーバヘッドの低下を実現する．

await?の性能を評価するため，ベンチマークの実行時間を標準の await を利用した場合と比較する実験を行なった．await?をすべての関数呼び出しに付与したと仮定してコード変換を行なったプログラムと，await をすべての関数呼び出しに付与したプログラムを用意して実行時間を計測した．

2　JavaScirpt における await

JavaScript と非同期処理は不可分の関係にある．JavaScript は Web アプリケーションを記述する際によく用いられており，ほとんどのブラウザがその実行エンジンを備えている．このような利用シーンでは，HTTP リクエストのような結果の返却に時間がかかる処理がしばしば登場する．処理の終了を待つ間，ブ

Toward a JavaScript Implementation of await with Low Overhead when Unnecessary.

Satoshi Kawamukai, Tetsuro Yamazaki, Shigeru Chiba, 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.

```
1 async function forgotten_await() {
2   const { request } = require("urllib")
3   const { data, res } = request("https://example.com/")
4   console.log(res.statusCode, data.length)
5 }
6 // TypeError: Cannot read properties of undefined (reading 'statusCode')
```

プログラム 1　await の書き忘れによるバグを含む関数と呼び出した際のエラーメッセージ

ラウザのレンダリングといった他の処理が停止するのを避けるため，時間のかかる処理は非同期的に実行されるように記述する必要がある．

JavaScript では，非同期処理を扱うために Promise [1] というオブジェクトが用いられている．JavaScript で非同期処理を記述する方法としては，Promise オブジェクトを明示的に利用する方法や，async/await 構文を用いる方法がある．Promise オブジェクトは非同期処理の成功，失敗，未完了を表す 3 つの状態を持ち，Promise オブジェクトには非同期処理の完了後に呼び出すコールバック関数を登録できるメソッドが用意されている．例えば then メソッドを用いると，処理の成功と失敗それぞれに対してコールバック関数を登録できる．then メソッドもまた Promise オブジェクトを返すため，更にコールバック関数を追加することが可能であり，これは Promise チェーンと呼ばれる．async/await は Promise オブジェクトを簡単に扱うための構文である．async キーワードを付与された関数 (以降は async 関数と呼ぶ) の中で，Promise オブジェクトを返す式に await を付与することで，Promise チェーンを明示的に構築することなく非同期処理を扱うことができる．async 関数は，最終的に Promise オブジェクトを返す．以降は，async 関数を含め Promise オブジェクトを返す関数を非同期関数，Promise オブジェクトを返さない関数を同期関数と呼ぶ．

async/await 構文を利用したプログラムにおけるバグの典型的な原因として，await を書き忘れることが挙げられる．プログラム 1 にその一例を示した．プログラム 1 では，HTTP リクエストのレスポンスを受け取って，ステータスコードを返却する関数を定義しているが，request メソッドの呼び出しの前に await を書き忘れてしまっている．await を書き忘れたことで，request メソッドが返却した Promise オブジェクトは，そのまま変数 data，res に分割代入される．その結果，data と res はともに undefined であるのに，res の存在しないプロパティを読み取ろうとしてエラーが発生する．このように，await の書き忘れによるバグは，エラーメッセージを見てもその原因に気づきにくいことがしばしばある．また型推論によって， ある程度 await の書き忘れを静的に発見できるが，呼び出される関数の本体を静的に決定できる場合など，発見可能な場合は限られる．

async 関数の中では，Promise オブジェクトを返さない式にも await を付与することが可能であり，await を付与した場合でも付与しなかった場合でも同じ結果が得られる．これを利用して，すべての関数呼び出しに保守的に await を付与することで，await を書き忘れることによるバグへの対処が可能と考えられる．

しかし，予期せぬバグを避けるためすべての関数呼び出しに付与するといった使い方を考えたとき，標準の await では同期関数の呼び出しに付与した場合の実行時オーバヘッドが大きい．詳細は 4 章の実験に譲るが，同期関数の呼び出し式のような Promise オブジェクトを返さない式に await を付与すると，プログラムの実行時間が実際に大幅に増加する．同期関数の呼び出しに await を付与する必要は本来なく，そのような await の使われ方を await の実装者が想定していないからだと思われる．現状の await の実装のままでは，await の使い方に制限が課されてしまっているといえる．

3　不要な場合でも速度低下が小さい await

本章では，新しい演算子として await?を提案する．この await?は標準の await と同じ動作をするが，

```
async function sample(url) {
    let conn = await? connectDB(url)
    let content = await? read(conn)
    return content
}
```

プログラム 2　コード変換の対象となるプログラム

```
function sample(url) {
    let conn = await? connectDB(url)
    let content = await? read(conn)
    return content
}
```

プログラム 3　async の削除

Promise オブジェクトを返さない式に付与された場合の速度低下が小さい．await?の特徴は，予め 2 つのパスを用意しておき，実行時に適切なパスを選んで処理を分岐させることにある．Promise オブジェクトを返さない式に付与されていた場合は，実行時間の短いファストパスへ分岐することでオーバヘッドを抑える．ファストパスは式が Promise オブジェクトを返さない前提で最適化されており，標準の await 相当の処理を省略した処理を行なう．一方，スローパスは式が Promise オブジェクトを返す前提で，標準の await 相当の処理を行なう．

3.1　コード変換の流れ

await?を JavaScript 上で実現するためのコード変換について述べる．なお，変換は関数単位で行う．実行時に適切なパスを選ぶために，await?が付与された式の返す値が Promise オブジェクトか否かでスローパスとファストパスに分岐する if 文を挿入する．スローパスには Promise オブジェクトの then メソッドに await?式以降の継続を渡す処理を記述する．一方，ファストパスには await?式以降の継続をそのまま実行する処理を記述する．継続を切り出すにあたっては，継続渡しスタイル（Continuation Passing Style, CPS）[5] への変換 (以降は CPS 変換と呼ぶ．) を行う．await?を for 文や while 文の中で利用した場合，この CPS 変換が複雑になり，かえって実行速度が低下するため，そのような場合は単に await?を標準の await で置き換えて変換を終了する．

```
function sample(url) {
    const kont1 = (v1) => {
        let conn = v1
        let content = await? read(conn)
        return content
    }
    return kont1(await? connectDB(url))
}
```

プログラム 4　継続の関数化

コード変換全体の手順は以下の通りである．以降でコード変換の対象となる await?が使用されたプログラムの例をプログラム 2 に示す．

1. ループの中に await?が含まれる場合，関数内のすべての await?を標準の await で置き換え，変換を終了する．
2. 関数宣言文から async キーワードを取り除く．
3. 実行順で最初に登場する await?式を変換の対象に選び，選択した await?式以降の継続を関数化する．
4. await?が付与された式の結果を変数に束縛する代入文を挿入する．
5. await?の対象である式の結果が Promise オブジェクトかどうかによって分岐する if 文を挿入する．
6. スローパスとファストパスを展開する．
7. 手順 3 で切り出した継続が await?を含む限り再帰的に処理する．

3.2　コード変換する関数の限定

手順 1 では，await?の使用に適さない関数を変換対象から除外している．await?を for 文や while 文の中で使用すると，手順 3 で継続を切り出す際の CPS 変換において，繰り返し文が再帰関数の呼び出しに変換されてしまうため，かえって実行速度の低下を招く．したがって，ループを含む関数は await?を標準の await で置き換え，async 関数としたまま変換を終了する．

3.3　await?式の選択と継続の切り出し

手順 3 では，実行順で最初に登場する await?式を

```
function anomal_sample(condition) {
  if(condition){
    await? a()
  }
  else{
    await? b()
  }
  c()
}
```

プログラム 5　条件分岐の節に await?が含まれる場合 (変換前)

```
function anomal_sample(condition) {
  if(condition){
    await? a()
    c()
  }
  else{
    await? b()
    c()
  }
}
```

プログラム 6　条件分岐の節に await?が含まれる場合 (変換後)

変換の対象に選び，その await?式以降の継続を関数化する．切り出した継続は，スローパスやファストパスの構築に利用する．プログラム 4 に，手順 3 を施した時点でのプログラムを示した．

プログラム 5 のように，await?が条件分岐の節の中に含まれており，それ以降で節の外側に処理が記述されている場合は特別な対応が必要となる．適切に継続を関数化するため，事前に節の外で行われる処理を条件分岐の各節の末尾に追加した上で手順 3 を行う．プログラム 5 に事前の変換を施したプログラムをプログラム 6 に示した．

3.4　代入文の挿入

手順 4 では，2 つのパスを設ける前の準備として，変換対象の await?が付与された式の結果を変数に束縛する代入文を挿入する．変換対象の await?が付与された式がもともと変数に代入されているのであれば，プログラムに書かれていた変数名をそのまま利用する．プログラム 7 に，手順 4 を施した時点でのプログラムを示した．

```
function sample(url) {
    const kont1 = (v1) => {
        let conn = v1
        let content = await? read(conn)
        return content
    }
    const v1 = connectDB(url)
    return kont1(await? v1)
}
```

プログラム 7　代入文の挿入

```
function sample(url) {
    const kont1 = (v1) => {
        let conn = v1
        let content = await? read(conn)
        return content
    }
    const v1 = connectDB(url)
    if (v1 instanceof Promise) {
        // スローパス
        return kont1(await? v1)
    } else {
        // ファストパス
        return kont1(await? v1)
    }
}
```

プログラム 8　パスを分岐させる if 文の挿入

```
function sample(url) {
    const kont1 = (v1) => {
        let conn = v1
        let content = await? read(v1)
        return content
    }
    const v1 = connectDB(url)
    if (v1 instanceof Promise) {
        return v1.then(kont1) // スローパス
    } else {
        return kont1(v1) // ファストパス
    }
}
```

プログラム 9　スローパスとファストパスの展開

3.5　スローパスとファストパスの追加

手順 5 では，await?が付与された式が Promise オブジェクトを返すかそれ以外の値を返すかによってファストパスかスローパスかに分岐する if 文を挿入する．instanceof 演算子を用いて変数の値を評価し，Promise オブジェクトであればスローパスへ，そうでなければファストパスへ分岐する．プログラム 8 に，手順 5 を施した時点でのプログラムを示した．

```
function sample(url) {
    const kont1 = (v1) => {
        const kont2 = (v2) => {
            let content = v2
            return content
        }
        let conn = v1
        const v2 = read(conn)
        if (v2 instanceof Promise) {
            return v2.then(kont2)
        } else {
            return kont2(v2)
        }
    }
    const v1 = connect(url)
    if (v1 instanceof Promise) {
        return v1.then(kont1)
    } else {
        return kont1(v1)
    }
}
```

プログラム 10　プログラム 2 の変換結果

手順 6 では，if 文の then 節と else 節の中にそれぞれスローパスとファストパスを展開する．スローパスが実行されるとき，変換対象の await?が付与された式は Promise オブジェクトを返したことになる．この Promise オブジェクトの then メソッドを呼び出し，手順 3 で関数化した継続を渡す．ファストパスが実行されるとき，変換対象の await?が付与された式は Promise オブジェクト以外の値を返したことになる．よって特別な対応は必要なく，手順 3 で関数化した継続に変換対象の await?が付与された式の結果を渡して実行する．プログラム 9 に，手順 6 を施した時点でのプログラムを示した．

3.6　再帰的な変換

手順 7 では，手順 3 で関数化した継続に対し，再帰的に変換を施す．プログラム 2 に対し以上の変換を施すと，最終的にプログラム 10 が得られる．

4　実験

標準の await と 3 章で提案した await?の性能を評価するため，3 つの実験を行なった．まず，(1) 標準の await を Promise オブジェクトを返さない式に付与した場合と付与しなかった場合の実行時間を比較する実験を行なった．3 章で提案した await?は標準の await と比較して，Promise オブジェクトが返却されなかった場合はより高速に動作し，Promise オブジェクトが返却された場合にも大きなオーバヘッドは生じないことが期待される．このことを確認するために await? を (2) Promise オブジェクトを返さない式に付与した場合と (3) Promise オブジェクトを返す式に付与した場合の両方について，標準の await を利用した場合とベンチマークの実行時間を比較する実験を行なった．

実験には，クロック周波数 3.2GHz，コア数 8 の CPU を持つ Apple M1 Pro と 16GB の LPDDR5 を搭載した PC を用い，プログラムはすべて Node.js 上で実行した．Node.js のバージョンは v18.16.1 であった．

4.1　ベンチマーク

同期的な関数呼び出しに await を付与したベンチマークは我々が知る限り存在しないため，実験のためにベンチマークプログラムを開発した．このベンチマークプログラムは Marr らの Cross-Language Compiler Benchmarking [2] に含まれるマイクロベンチマークを元に，関数宣言文に async を付与し，その内部の関数呼び出し式に await? ないし await を付与することで作成したまた各関数が Promise オブジェクトを返却するかどうかを後から制御できるように，関数呼び出しを別の関数でラップし，フラグに応じて戻り値を Promise オブジェクトで包む条件分岐を加えた．

await? ないし await の挿入はソースコード変換器によって機械的に挿入し，その後，手動で調整を加えた．このソースコード変換器は JavaScript トランスパイラの Babel を利用して開発した．await? を含むベンチマークはそのままでは実行できないため，3 章で示した方法で変換してから実行し，実行時間を計測した．

また，ベンチマークプログラムの中でも全ての関数で await? が for 文や while 文の中で出現するものは除外した．このような関数では全ての await? が変換手順 1 で標準の await に置き換えられてしまい，await? か await かに関わらず同じプログラム

表 1　ベンチマークの実行時間 [ms]

ベンチマーク	Promise を返す？	await?	await	await 無し	実行時間の比 await?/await
Towers	No	29.19	184.67	8.55	**15.8%**
List	No	13.91	42.66	3.95	**32.6%**
Storage	No	78.20	79.95	9.78	97.8%
Bounce	No	45.67	42.02	3.20	108.7%
Towers	Yes	303.26	317.51	–	95.5%
List	Yes	73.46	66.26	–	110.9%
Storage	Yes	128.53	128.29	–	100.2%
Bounce	Yes	72.11	68.36	–	105.5%

になってしまうためである．

4.2　実験結果

表 1 に各ベンチマークに対する実行時間を示した．

(1) 標準の await を Promise オブジェクトを返さない式に付与した場合，付与しなかった場合と比較してベンチマークの実行時間は 8 倍から 22 倍程度に増加した．Promise オブジェクトを返さない式に本来不要であるはずの await を付与した場合，大きな速度低下がみられる．

(2) Promise オブジェクトを返さない式に付与した場合，標準の await との比較では，await? はベンチマークによって実行時間が 15.8% と大幅に短縮されるものから実行時間が 108.7% に増加してしまうものまでばらつきのある結果であった．このばらつきの原因は，ベンチマークの中にループを含む関数が多いほど，await?を利用しコード変換を施したプログラムに，標準の await を付与したプログラムとの共通部分が増えるためであると考えられる．例えばループを含まず再帰関数によって計算を行う List および Towers では await?を付与したことによる実行時間の大きな改善が見られたのに対して，関数呼び出しの大半が for 文の中にある Bounce や Storage では改善はみられなかった．

また await 無しとの比較では，3.5 倍から 14 倍程度の速度低下が確認された．関数がループを含まない List や Towers ではどちらも 3.5 倍程度の性能低下であるため，同期処理に対するファストパスのオーバヘッドはこの程度であると考えられる．

(3) Promise オブジェクトを返す式に付与した場合，標準の await と比較した await? の実行時間のオーバヘッドは -4.5% から +10.9% と，大きな差は見られなかった．これは，await?を利用した場合でも await を利用した場合でも Promise オブジェクトを処理しているためであると考えられる．

5　関連研究

我々の研究は，不足している await を静的に検出するのではなく，バグを避ける目的で余分に付与しても実行速度の低下が起きにくい await の実装方法を提案する点が新しい点である．非同期処理に由来するバグの静的解析に関する手法はいくつか提案されている．

Sotiropoulos らは JavaScript の非同期処理の静的解析のために非同期処理の実行順序を解析する手法を提案している [4]．この手法では，既に提案されていた JavaScript の計算モデルに非同期処理に関連する部分を追加し，コールバック関数の実行順序を有向非巡回グラフとして表現する．中規模のベンチマークを用いた性能評価実験では，プログラム中の非同期処理同士の実行順序を平均で 79% 決定することができる．この手法は，async/await を計算モデルに組み込んでおらず，また静的解析ツールの開発にまでは至っていない．

Rau らは Promise を利用するプログラミングのために線形型システムを組み込んだ新たな言語を考案した[3]．Rau らはケーススタディとして，`Promise` オブジェクトの操作に関連するバグを含む JavaScript のプログラムを提案した言語に変換し，頻出するバグが発見できることを示している．しかし，例外処理を含むものなど提案した言語に変換できなかったプログラムが多く，また `async`/`await` におけるバグについては扱われていない．

6 まとめ

本論文では，保守的に `Promise` オブジェクトを返さない式に `await` を利用する場合を想定して，その際のオーバヘッドが `await` より小さい `await?`と，それを JavaScript 上で実現するためのコード変換を提案した．`await?`と標準の `await` を比較する実験を行い，不要な場所に書かれた場合の実行時間が短縮され, 必要な場所に書かれた場合には標準の `await` と同等の性能を示すことを確認した．

参 考 文 献

[1] Friedman, D. and Wise, D.: *The Impact of Applicative Programming on Multiprocessing*, Technical report (Indiana University, Bloomington. Computer Science Department), Indiana University, Computer Science Department, 1976.

[2] Marr, S., Daloze, B., and Mössenböck, H.: Cross-Language Compiler Benchmarking: Are We Fast Yet?, *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, Association for Computing Machinery, 2016, pp. 120–131.

[3] 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, Dagstuhl, Germany, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, pp. 13:1–13:27.

[4] Sotiropoulos, T. and Livshits, B.: Static Analysis for Asynchronous JavaScript Programs, 2019.

[5] Sussman, G. and Steele, G.: Scheme: A Interpreter for Extended Lambda Calculus, *Higher-Order and Symbolic Computation*, Vol. 11(1998), pp. 405–439.