6J-05

Liquid: 非同期一階関数による並行計算体系*

諏訪重貴† 福田浩章‡ 篠埜功§
芝浦工業大学¶

概 要

ウェブアプリケーションのようなソフトウェア開発において，非同期プログラミングを用いた並列実行が必須である．非同期処理を扱うモデルである Callback，Thread，Promise[1]，Future[2, 3]，Async/Await[4] 等を用いる場合，非同期処理に対する操作を明示的に記述する必要があり，プログラムの複雑さが増加する要因となる．非同期処理と同期処理の区別なくコードを記述することが可能にすることで，プログラムの保守性・可読性向上させ致命的なバグの排除に繋がると考えられる．これを実現するため，本研究では Liquid という計算体系を提案し操作的意味論を定義した．

1. 既存技術の問題点

C や Java で通常用いられる同期処理は，プログラムの記述順に処理が実行される．そのため，ハードディスクやネットワーク等の時間のかかる入出力処理でも，その処理の完了までプログラムの実行が一時停止することになる．SPA(Single Page Application) として構築されるようなウェブアプリケーションでは，画面の描画と並行してサーバとの通信を実行するため，非同期処理を用いることが必須である．しかし，非同期処理を用いるプログラムは同期処理のものと比較して複雑であり，プログラムの可読性や保守性が低下する可能性がある．本章では，非同期処理の実装に用いられる技術のうち，Callback，Promise，Async/Await を取り上げ，問題点を述べる．

1.1 Callback

Callback 関数を用いるプログラミングスタイルは Continuation Passing Style[5] と呼ばれ，古くから存在する手法である．これを非同期処理に用いることで，プログラム全体の実行を停止せずに並列実行することが可能になる．Callback 関数を用いた非同期処理のコード例を 1 に示す．入出力処理などの実行時間が長い関数 (httpRequest) は，関数 (これを Callback 関数と呼ぶ) を引数として受け取る．Callback 関数の内部では，httpRequest の処理結果 (response) を用いることができる．しかしこの手法を用いて複数の非同期処理を順番に実行する場合，Callback 関数の入れ子が深くなる Callback Hell[6] と呼ばれる問題が起きやすく，可読性や保守性を損なうことがある．

*Liquid: A concurrent calculus with declaring first-order asynchronous functions
†Shigeki Suwa
‡Hiroaki Fukuda
§Isao Sasano
¶Shibaura Institute of Technology

Listing 1: Callback 関数を用いたコード例（JavaScript）

```
1 function myTask() {
2   httpRequest(function(response) {
3     print(response);
4   });
5 }
```

1.2 Promise

Promise は，前述した Callback を用いた非同期処理を抽象化したクラスである．Promise を用いることで，連続した非同期処理をメソッドチェーン形式で記述することができ，Callback Hell の問題を回避することができる．Promise は非同期処理の抽象化モデルとしては有効であるが，プログラマは特殊なプログラミングスタイルを要求される．よって，プログラマの学習コストが高く，効率的なコードを書くにはある程度の習熟度が必要となる．

1.3 Async/Await

Async/Await は非同期処理を扱うための構文であり．C#，JavaScript，Python 等で利用可能である．Listing2 は，Async/Await を用いて Listing1 と同じ振る舞いをするように書き換えたコードである．async 修飾子を用いて定義された関数 (myTask) は async 関数と呼ばれ，その関数本体で await キーワードが利用可能になる．await が適用された部分以降の処理は Continuation としてキャプチャーされ，httpRequest で実行される非同期処理の完了後に呼び出される．また，await が適用された部分で async 関数の実行は一時中断し，プログラムの実行は呼び出し側へ戻るため，プログラム全体の実行は停止されない．

Listing 2: Async/Await を用いたコード例（JavaScript）

```
1 async function myTask() {
2   var response = await httpRequest();
3   print(response);
4 }
```

2. Liquid

これまで述べたモデルを利用して，非同期処理を実行する関数を呼び出すとき，それぞれのモデルや概念を理解した上で，処理の実行順に配慮しながら，適切にコーディングする必要があるため，プログラマの学習コストやコーディングの難易度が高い．したがって，非同期処理と同期処理の区別なくコードを記述することが可能になれば，プログラムの保守性・可読性向上に寄与できると考えられる．

そこで本論文では，この目的を達成する計算体系 Liquid を提案する．本節では，Liquid の振る舞い，構文，

$$
\begin{array}{rcl}
prog & := & decls\ t \qquad decls := \epsilon \mid decl\ decls \\
decl & := & \textbf{fun}\ id\ p = t \mid \textbf{async fun}\ id\ p = t \\
p & := & id \mid (id,\ id) \\
t & := & id \mid id\ t \mid id\ (t_1,\ t_2) \\
 & \mid & \textbf{and}\ (t_1,\ t_2) \mid \textbf{not}\ t \mid \textbf{true} \mid \textbf{false}
\end{array}
$$

図 1: Liquid の構文定義

コンパイル，意味論の定義の概要について述べる．

2.1 Liquid の振る舞い

Liquid が実現したいプログラムの動作について，Listing3 に示す疑似コード用いて述べる．

まず，非同期で実行する処理を async キーワードを用いて非同期関数 (request) として宣言する（1-3 行目）．この非同期関数の呼び出し（4 行目）は，その処理の完了を待たずに終了し，次の行の実行を試みる．非同期関数の戻り値 (response) を用いた関数呼び出し（5 行目）は，その非同期処理が完了した後に実行するため，この時点では実行をスキップし，次の行の実行を試みる．非同期関数の戻り値と関係ない処理（6 行目）は，非同期処理の完了を待たずに実行する．最後に，先に実行していた非同期処理が完了すると，スキップした処理（5 行目）を再開する．

以上の振る舞いによって，同期関数の呼び出しと同様の記述を保ち，非同期処理の実行と結果の取得を記述することが可能になる．

Listing 3: Liquid の擬似コード

```
1 async function request() {
2   ...
3 }
4 var response = request();
5 print(response);
6 anotherTask();
```

2.2 Liquid の構文定義とコンパイル

Liquid の構文定義を図 1 に示す．Liquid のプログラムは任意数の一階関数の宣言に続けてプログラム本体を記述する．async を用いて宣言された関数は非同期関数となる．この構文で記述されたプログラムは，core Liquid にコンパイルしてから評価（実行）する．コンパイラは，非同期関数として宣言された関数呼び出し部分が future 構文で囲われるように変換する．これは，後述する core Liquid の意味論の定義によって非同期で実行されるようにするためである．

2.3 core Liquid

core Liquid の操作的意味論の定義を図 2 に示す．t は core Liquid の部分式，a は future 構文を含む部分式，v は値，id は関数名を表し，$id\ (t_1,\ t_2)$ は関数適用式を表す．core Liquid は，関数の区別（非同期関数かどうか）ではなく future 構文によって非同期実行であることが明示される．future 構文は (E-FUTURE-1) と (E-FUTURE-2) によって評価される．また，(E-ARG-1)，(E-ARG-2)，(E-ARG-3) よって関数適用式の引数の評価順序に自由度が発生する．これによって，第 2.1 節で述べた振る舞いを実現する．

$$
\text{(E-FUTURE-1)}\ \frac{E \vdash t \to t'}{E \vdash \textbf{future}\ t \to \textbf{future}\ t'}
$$

$$
\text{(E-FUTURE-2)}\ \ E \vdash \textbf{future}\ v \to v
$$

$$
\text{(E-ARG-1)}\ \frac{E \vdash t_1 \to t_1'}{E \vdash id\ (t_1,\ t_2) \to id\ (t_1',\ t_2)}
$$

$$
\text{(E-ARG-2)}\ \frac{E \vdash t_2 \to t_2'}{E \vdash id\ (a_1,\ t_2) \to id\ (a_1,\ t_2')}
$$

$$
\text{(E-ARG-3)}\ \frac{E \vdash t_2 \to t_2'}{E \vdash id\ (v_1,\ t_2) \to id\ (v_1,\ t_2')}
$$

図 2: core Liquid の評価規則（一部）

3. まとめ

非同期処理は様々な状況で用いられ，ウェブアプリケーションのようなソフトウェア開発では必要不可欠となっている．本論文では，非同期処理を扱うモデルの問題点を挙げ，それを解決するための計算体系 Liquid を提案した．また，Liquid の振る舞い，構文，コンパイル，意味論の定義の概要について述べた．Liquid を用いることで，非同期処理と同期処理の区別なくコードを記述することが可能になり，プログラムの保守性・可読性向上が期待できる．今後の課題として，高階関数の導入と，それに伴い非同期関数型を表す型システムの導入などが挙げられる．

参考文献

[1] Ecma International. *ECMAScript®2015 Language Specification.* http://www.ecma-international.org/ecma-262/6.0/ 2018/01/09 参照.

[2] Robert H. Halstead, Jr. *Multilisp: a language for concurrent symbolic computation.* ACM Transactions on Programming Languages and Systems. 1985.

[3] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. *Futures and promises.* http://docs.scala-lang.org/overviews/core/futures.html 2018/01/09 参照.

[4] Nick Benton, Luca Cardelli, and Cdric Fournet. *Modern concurrency abstractions for C#.* European Conference on Object-Oriented Programming. Springer. 2002.

[5] A. W. Appel and T. Jim. *Continuation-passing, closure-passing style.* In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1989.

[6] *Callback Hell.* http://callbackhell.com/ 2018/01/09 参照.