5M-04

Liquid: 非同期処理の暗黙的同期を行う型付きプログラミング言語*

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，ネットワーク等の入出力を非同期処理として実装することは必要不可欠である．非同期処理のためのモデルを用いる場合，並行性を制御するための明示的な関数呼び出しやキーワードの適用，そして同期するタイミングの注意が求められる．これは，非同期処理を含んだプログラムがそうでないプログラムと比べて複雑であると言える．非同期処理のための特別な記述を用いることなくプログラムを記述可能にすることで，プログラムの保守性・可読性を向上させ致命的なバグの発生を抑制できると考えられる．本論文では，これを実現する Liquid という計算体型を定義し，その計算体系を実現する処理系を実装する．

1. 非同期処理のモデル

ネットワーク等の入出力処理を同期処理として実行した場合，処理の完了までプログラムの実行は一時停止する．これを非同期処理に置き換えることでプログラムが停止せずに他の処理を実行できるため，ウェブアプリケーション等に多く用いられる．非同期処理を用いる場合，並列性を制御するため通常の同期プログラミングにはない考え方やプログラミング手法を必要とし，非同期処理を複雑なものにしている．本節では非同期処理に用いられるいくつかのモデルについて述べる．

1.1 Callback

Continuation Passing Style [1] で用いられる Callback の仕組みを，非同期処理の制御に応用することができる．Listing 1 に示すように，ある非同期処理に対して，その完了後に実行したい一連の処理を Callback 関数 (2 行目) として登録することで非同期処理終了後の処理を記述できる．

この手法による非同期処理の実装は，Callback Hell [2] の問題が存在し，プログラムの可読性を下げる原因となる．また，非同期処理の実行タイミングに依存する処理は Callback 関数内に，そうでない処理は Callback 関数外に分けて記述することになり，プログラマは実行タイミングへの注意が必要である．

Listing 1: Callback による非同期処理

```
1 let myFunc = function() {
2     httpReq("example.com", res => print(res));
3 }
```

Listing 2: Promise による非同期処理

```
1 let myFunc = function() {
2     let promise = httpReq("example.com");
3     promise.then(res => print(res));
4 }
```

Listing 3: async/await による非同期処理

```
1 let myFunc = async function() {
2     let res = await httpReq("example.com");
3     // 以降ブロックの影響を受ける
4     print(res);
5 }
```

1.2 Promise

非同期処理を抽象化するモデルである Promise [3] は，JavaScript，C#，Scala 等で導入されている．ライブラリ関数等によって，非同期処理の抽象表現となる Promise オブジェクトが生成される．Listing 2 に示すように，Promise が有する then メソッド等によって，その非同期処理に対する Callback 関数を登録 (3 行目) することができる．

Promise によって Callback Hell の問題を回避できるが，Callback 関数内外の実行タイミングへの注意は変わらず必要である．

1.3 Future, async/await

Future は，Multilisp [4] をはじめ，Alice [5] ，C++，Java，Scala 等で導入されている概念である [3] ．非同期処理の結果得られる値を Future オブジェクトとして一時的に表現し，Future が有する get メソッド等の呼び出しや Future 自身が評価の対象となったとき，実際の結果を取得する．そのとき非同期処理が完了するまでブロックし，以降に記述された処理はその後に実行される．

一方 JavaScript や C#では，Listing 3 に示すように async 関数の中で Promise に対して await キーワードを適用 (2 行目) できる．非同期処理の結果を戻り値として取り出すことができ，より簡潔な記述になる．

しかし，Future ではそのスレッドが，async/await ではその関数及び Promise オブジェクトが長時間ブロックする可能性があり，プログラマは適切なタイミングでブロックが発生するよう注意が必要である．

2. Liquid

これまで述べた非同期処理のためのモデルでは，並行性を制御するための明示的な関数呼び出しやキーワードの適用，そして同期のタイミングへの注意が必要である．これらは通常の同期プログラミングにはない考え方であ

*Liquid: A typed programming language with implicit synchronization for asynchronous execution
†Shigeki Suwa
‡Hiroaki Fukuda
§Isao Sasano
¶Shibaura Institute of Technology

Listing 4: Liquid の擬似言語によるプログラム

```
1 let httpReq = (async fun url: String =
2     /* call primitive */ ) in
3 let anotherTask = (fun flag: Bool =
4     /* ... */ ) in
5 let res = httpReq "example.com" in
6 (print res,
7     anotherTask true)
```

り，非同期処理を複雑なものにしている．本節では，これらを極力必要としない計算体系 Liquid を定義し，その計算体系を実現する処理系の実装について述べる．

2.1 Liquid のコンセプト

Liquid のコード例を Listing 4 に示す．基本的に命令は逐次実行，評価戦略は値呼びである．非同期関数 (1-2 行目) と通常の関数 (3-4 行目) を定義することができ，非同期関数は並行実行する．非同期関数の呼び出し (5 行目) 自体は即時終了し，戻り値は Future オブジェクトとなる．その Future オブジェクトを必要とする命令 (6 行目) は，非同期関数の実行終了後に Future オブジェクトを実際の値に置き換えて実行する．そうでない命令 (7 行目) は，非同期関数にブロックされることなく，6 行目よりも先に実行される．

Liquid は，get メソッドや await キーワード等の明示的な命令が不要で，かつ命令の記述順に関わらず効率的な非同期処理を実装することができる．

2.2 Liquid の定義とコンパイル

Liquid の構文定義を図 1 に示す．Liquid は，真偽値，ペア，関数宣言・適用等の最小限の要素を持つ関数型言語である．Liquid のプログラムは型検査後，後述する core Liquid にコンパイルされ，意味論に従って評価 (実行) される．コンパイラは，非同期関数の関数適用を `future` で囲うように変換する．型 $T \xrightarrow{a} T$ が非同期関数を表すため，高階関数の引数や戻り値にも非同期関数を取ることができる．型付け規則とコンパイルアルゴリズムについては省略する．

2.3 core Liquid の意味論

core Liquid の操作的意味論の一部を図 2 に示す．t は項，a は `future` 項を子孫に含む項，v は値である．core Liquid は $(a,\ t)$ のような項に対して複数の評価規則を適用できる．これは，評価順に自由度を持たせることで，`future` 項の評価と同時に他の部分を並行に

$$
\begin{aligned}
v &:= \mathbf{true} \mid \mathbf{false} \mid (v,\ v) \\
t &:= v \mid x \mid p \mid (t,\ t) \mid t.1 \mid t.2 \mid t\ t \\
&\quad \mid \mathbf{fun}\ x : T = t \mid \mathbf{async\ fun}\ x : T = t \\
&\quad \mid \mathbf{let}\ x = t\ \mathbf{in}\ t \mid \mathbf{not}\ t \mid \mathbf{and}\ t \\
T &:= \mathbf{Bool} \mid (T,\ T) \mid T \to T \mid T \xrightarrow{a} T
\end{aligned}
$$

図 1: Liquid の構文定義

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

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

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

図 2: core Liquid の評価規則 (一部)

評価できることを表す．Listing 4 において，(`print res, anotherTask` **true**) は，`res` が `future` 項となり，`print res` と `anotherTask` **true** のどちらも評価できる．

2.4 Liquid の実装と評価

Liquid を型検査，コンパイル，解釈実行する，インタプリタを JavaScript で実装した[1]．`future` 項を評価すると Promise オブジェクトが生成される．ペアや関数適用を評価するとき，子要素に持つ Promise オブジェクトの完了後に実際の評価を実行するような新しい Promise オブジェクトを生成することで，`future` 項に関する振る舞いを実現する．

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，従来の非同期処理のモデルが持つ複雑さのない非同期プログラミングを可能にする言語 Liquid を定義した．Liquid は非同期処理の実行や結果の取得を同期処理であるかのように記述でき，プログラムの保守性・可読性の向上や致命的なバグの発生を抑制することなどが期待できる．

今後の課題として，より実用的に Liquid の特徴を利用可能にするため，TypeScript を拡張した AltJS 等既存の言語を拡張するような実装を考えている．

参考文献

[1] 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.

[2] Max Ogden. *Callback Hell.* http://callbackhell.com/ 2018/12/19 参照.

[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 2018/12/19 参照.

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

[5] Joachim Niehren, Jan Schwinghammer, Gert Smolka. *A Concurrent Lambda Calculus with Futures* Theoretical Computer Science 364(3). 2006.

[1] `https://github.com/ztrehagem/liquid-alpha`