限定継続演算子の拡張および それを用いた並列向け領域特化言語群の実現

山口 洋　千葉 滋

本稿では，埋め込み領域特化言語の表現力を向上させるための言語拡張を提案する．この言語拡張は並列化向け領域特化言語群の実装に有効である．例えば，フォーク・ジョイン型の並列化プリミティブに対しては逐次と同様の構文を提供できるようになる．この構文は，メソッドやクロージャのような広く使われている言語機構のみを用いた場合には困難であったものである．さらに限定継続演算子と組み合わせたことにより，ノンブロッキング型の並列化プリミティブに対しても同様の領域特化言語を実現可能である．この領域特化言語群により，リファクタリングコストの削減と並列化の誤りの防止が期待される．また，提案する演算子の実装も試みた．

1　はじめに

本稿では，埋め込み領域特化言語 (DSL, Domain Specific Language) の表現力を向上させるための汎用の言語機構を提案する．ある領域に特化した言語プリミティブを構築することに使われる既存の言語機構としては，メソッド・クロージャや Scala の名前呼び引数，Ruby のブロック引数のように暗黙的にサンクを生成する機構などが挙げられる．しかしながら，しばしばこれら既存の言語機構だけでは十分なプリミティブを得られず，埋め込み DSL を諦めて，専用コンパイラや強力なマクロ機構による専用 DSL を頼らざるをえないことが多い．

既存の言語機構だけでは十分なプリミティブを得られない例は並列化プリミティブである．各並列化プリミティブは本質的に独自の構文を必要としており，既存の言語機構で構築可能な範囲では強い構文的な制約を受け，自由に便利な構文を提供することが難しい．実際，Cilk [5] [6] を始めとして数多くの並列向け DSL が提案されているが，その多くが専用のコンパイラを必要とする．

Hiroshi Yamaguchi, Shigeru Chiba, 東京大学情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

その中でも代表的な並列化プリミティブ群として，フォーク・ジョイン型を取り上げる．これは，計算の分岐を行うフォークと，フォークされた計算の結果を合流させるジョインの組からなるようなプリミティブである．このフォーク・ジョイン型プリミティブは，既存の言語機構で構築可能な範囲では構文的な制約が大きく，並列化前の逐次のプログラムの構造を壊さないように書き加えるという制限の下では，そのプログラムを並列化することができない．そのため，リファクタリングコストの増大や，並列化の誤りといった問題が生じる．

本稿では，新しい言語機構として，限定継続演算子に必要呼びのアイディアを加えた新しい演算子を提案し，この演算子によってその問題を解決する．また，フォーク・ジョイン型プリミティブには，一般的なブロッキング型だけでなく，ノンブロッキング型のプリミティブもあるが，それらのプリミティブに対しても本演算子は有効である．また，本稿では提案する演算子を Scala に対する言語拡張として実装したプロトタイプについても述べる．

2　領域特化言語における構文的な制約

既存の言語機構で実現できる範囲でフォーク・ジョイン型の並列化プリミティブを用いた領域特化言語

```
return (f() + g()) * v;
```

```
return (par(() -> f()) + g()) * v;
```

```
final Future<Integer> x =
  ForkJoinPool.commonPool().submit(() -> f());
final int y = g();
return (x.get() + y) * v;
```

図 1　フォーク・ジョイン型における構文的な制約

を設計する場合，構文的に制約を受けることがある．例えば図 1 の 1 番目のような，簡単な Java 8 のプログラムをフォーク・ジョイン型のプリミティブを用いて並列化することを考える．このプログラム中の `f()` をフォークしたいとする．元のプログラムの構造を保とうとするならば，図 1 の 2 番目のように式の一部を切り取って渡すことで並列化できるようなプリミティブが構築できると良い．ここで `par` は，サンクとしてラムダ式†1を渡すと何らかの適切な方法でフォークして並列化する仮想のプリミティブとする．しかしながら，この `par` は実現が困難である．Java 8 において，典型的なフォーク・ジョイン型の並列化プリミティブとしては，`java.util.concurrent` パッケージの `Future` がある．このプリミティブを用いたプログラムは，図 1 の 3 番目のように書かれ，1 番目とは構文的に大きく異なってくる．まず，1 個の式で書かれていたプログラムは $f()$，$g()$，$(f() + g()) \times v$ の 3 段階に分割される．その上で，2 行目の `ForkJoinPool.commonPool().submit` で $f()$ をフォークし，フォーク点とは離れた位置にある 4 行目の `get` でその計算結果をジョインさせる必要がある．

つまり，フォーク・ジョイン型において受ける構文的な制約とは，フォーク点とジョイン点を明示的に書く必要があるということである．前述の `par` ではフォーク点しか明示されず，ジョイン点を示すプリミティブがないため，一般的なフォーク・ジョイン型では実現できないプリミティブである．一般に，実現可能な，フォーク・ジョイン型を用いた領域特化言語は，必ず，フォーク点とジョイン点の両方を示すプリミティブが揃っている必要がある．

我々は，ジョインする点が暗黙的に補完されるような構文を，埋め込み領域特化言語でも実装可能にすべきと考えている．それによって，構文的な制約を軽減することができ，逐次のプログラムの構造を保った並列化が可能なプリミティブを実現できるからである．例えば，そのようなプリミティブ `par` はジョイン点を明示しないため実現できなかったが，ジョイン点の補完により実現可能なプリミティブとなる．

現実には暗黙的な補完を実現できる言語機構はなく，逐次のプログラムとの構文的な違いを十分に埋めることはできない．そのため，リファクタリングのコストが増大したり，誤りが起きやすいといった問題が生じる．

リファクタリングのコストが増大する例は，図 2 のようなフォークする範囲を変更する場合である．いずれも $(f() + g()) \times h()$ を計算するプログラムであるが，フォークする範囲が異なり，上は $f()$ をフォークし，下は $f() + g()$ をフォークする．このように言葉による説明の上では軽微な変更であるが，図に示した通り，実際のプログラムの上では大きな変更となり，プログラムのすべての行を修正する必要がある．特に注目すべき点は，3 行目の `y = g()` から `z = h()` への変更である．これは，フォークする範囲を変更するときには，同時に，フォークする計算と並列に行う計算も変更しなければならないからである．このように，フォーク・ジョイン型では並列化によってプログラムの構造が大きく変化するだけでなく，その構造はフォークする範囲によって大きく異なる．そのため，フォークする範囲の変更もまた，プログラムの構造を大きく変更することが必要となり，リファクタリングのコストが増大する，

間違いを起こす例としては，図 3 のようにジョイン点を誤って指定する場合が挙げられる．このプログラムは，逐次のプログラムの構造を壊さないように並列化しようとして失敗した例である．このプログラムは，Java 8 のプログラムとして正しいものであるが，フォークした点の直後にジョインする点があるた

†1　Java 8 で新たに導入された言語機構

```
final Future<Integer> x =
  ForkJoinPool.commonPool().submit(() -> f());
final int y = g();
return (x.get() + y) * h();
```

```
final Future<Integer> xy =
  ForkJoinPool.commonPool().submit(() -> f() + g());
final int z = h();
return xy.get() * z;
```

図 2　リファクタリングコストが高い例

```
return
  (ForkJoinPool.commonPool().submit(() -> f()).get()
    + g()) * v;
```

図 3　典型的な誤りの例

```
static R par<T, R>
  (final Supplier<T> thunk) {
    return tearOff(
      k -> k.apply(
        ForkJoinPool.commonPool().submit(thunk)),
      (x, k) -> k.apply(x.get())
    );
}
```

図 4　par の実装

め全く並列化できていない．フォーク・ジョイン型のプリミティブの構文は逐次の構文と酷似しているが，実際には構文的に大きな違いが存在している．そのため，プリミティブに対する理解が不十分であると，このように間違いが起こる．

3　提案: ticket 演算子

前節の問題を解決する言語拡張として ticket 演算子を提案する．Ticket 演算子は，限定継続演算子 *shift/reset* [2][1][9] に必要呼びの戦略を組み合わせて，暗黙的なジョイン点の補完を可能にした演算子である．これにより，例えば Java 8 では，`par` を図 4 のように実装できる．

Ticket 演算子は，拡張された継続演算子 *tearOff* と継続の切り取り範囲を制限する *reset* の 2 つの演算子の組からなる．このうち *reset* は *shift/reset* におけるそれと同一である．また，*tearOff* はこれまでの継続演算子とは異なり，2 段階に分けて継続を切り取ることで 2 種類の継続，すなわち従来の演算子における現在の継続に加えて “未来の継続” を同時に扱えるようになっている．

TearOff は 2 つの引数をとり，それぞれ現在の継続，未来の継続に相当するクロージャを受け取り何らかの処理をするクロージャである．本稿ではこれらのクロージャをコントローラと呼ぶことにする．図 4 においては，それぞれ，`k -> k.apply(...)` が現在の継続，`(x, k) -> k.apply(x.get())` が未来の継続に対するコントローラであり，継続は引数 `k` として渡される．

TearOff を実行すると次のように動作する．まず，現在の継続がクロージャとして切り取られ，対応するコントローラに渡される．次にコントローラ内で，その現在の継続に相当するクロージャが使われる．このとき，このクロージャは通常とは異なり，必要呼びのように動作する．まず，クロージャの引数が実際に必要になる直前まで実行される．すなわち，クロージャを `x -> {...; y = f(x); ...}` と模式的に表現すると，`y = f(x)` の直前まで実行され，そこで一旦停止するということである．そして，そこで 2 段目の継続の切り取りが行われ，それが未来の継続として対応するコントローラに渡される．つまり，*tearOff* は通常の現在の継続の切り取りに加えて，必要呼びのような戦略で追加の継続を切り取ることができる．

TearOff 演算子には他にも特徴的な点があり，それは，現在の継続の引数の型を変更し，さらにその引数を未来の継続のコントローラに追加の引数として渡すことができる点である．これにより，図 4 の 4–5 行目のように，*tearOff* 演算子で切り取った現在の継続に対して，サンクを実際に計算した値の代わりにフューチャーオブジェクトを渡すことができる．また，図 4 の 6 行目のように未来の継続のコントローラには 2 個の引数があるが，1 個目の引数 `x` はこの機能により追加された引数である．未来の継続のコントローラは，この引数を通して，4–5 行目で現在の継続に渡されたフューチャーオブジェクトを受け取ることができる．

```
static R nonblocking_par<T, R>
  (final Supplier<T> thunk) {
    return tearOff(
      k -> k.apply(
        CompletableFuture.supplyAsync(thunk)),
      (x, k) -> x.thenApplyAsync(k)
    );
}
```

図 5　ノンブロッキング型における par の実装

このように，2 つのコントローラ間で値の受け渡しに使用することができる．

また，フォーク・ジョイン型には，ここまでで取り上げた一般的なブロッキング型に加えてノンブロッキング型が知られている．ノンブロッキング型がよく見られる言語として node.js [7] があり，Java 8 においても `CompletableFuture` が存在する．ノンブロッキング型のプリミティブでも図 5 のようにすることで ticket 演算子により実現できる．一般には，ノンブロッキング型のようなモナドに類するデザインパターンに対しては継続演算子が有効であることが知られており [4]，ticket 演算子はその理論を拡張する形で設計されている．

4　実装と実験

提案する ticket 演算子が実際に機能するかを確認するためにプロトタイプの実装を行った．プロトタイプは Java 8 ではなく，Scala 2.11 の言語拡張として実装した．具体的には，Scala 2.11 標準の型付き構文マクロシステムを用いて，Scala に *shift/reset* を追加する CPS プラグインの実装 [8] を参考に，型主導 CPS (Continuation Passing Style, 継続渡し形式) 変換のアイディアに基づいて実装した．

また，簡単なパフォーマンス測定を行った．実験対象として用いたのは，*async/await* というプリミティブである．このプリミティブは，ノンブロッキングな並列化を行うマクロライブラリ Scala Async [3] によって提供される．これを用いてフィボナッチ数の計算を行った．比較対象として，逐次のままのプログラムと，Scala Async をそのまま用いたプログラム，*shift/reset* を用いて Scala Async に似せて愚直に実装

```
def fib(n: Int): Future[Int] = async {
  if (n < 2)
    1
  else {
    val xf: Future[Int] = fib(n - 1)
    val yf: Future[Int] = fib(n - 2)
    await(xf) + await(yf)
  }
}
```

```
def fib(n: Int): Future[Int] = async {
  if (n < 2)
    1
  else
    await(fib(n - 1)) + await(fib(n - 2))
}
```

図 6　2 つの領域特化言語によるフィボナッチ数の計算

したプリミティブによるプログラム，提案した ticket 演算子を用いて実装したプリミティブによるプログラムの 4 種類を用いた．元の Scala Async はマクロによって実装されているため，*shift/reset* 版より高速であると予想された．また，Scala Async によるプログラム，*shift/reset* によるプログラムは同一であり，それを図 6 上に，ticket 演算子によるプログラムを図 6 下に示す．Ticket 演算子を使用すると逐次版のプログラムと同様の構造になる．

実験環境は，プロセッサが Intel Core i7-4770S 3.10GHz で 4 コア 8 スレッド，Java VM は OpenJDK 1.7.0_51 64 ビット版, Scala コンパイラは 2.11.1 で, 最適化オプションとして “`-optimise`” のほか 5 種類†2 を使用した．計測は，JIT の影響を抑えるため最初の 10 回を捨て，その後の 20 回の中央値をとった．

それぞれのプログラムについて *fib*(30) を計算した結果は表 1 の通りである．Ticket 演算子を使用して実装したプリミティブが動作すること自体は確認できるものの，*shift/reset* を使用して実装したプリミティブと比較すると並列化にかかったオーバーヘッドが 3 倍，Scala Async そのものと比較すると 10 倍と非常に大きい．これは継続をラッピングするための中間オブジェクトのコストがかかること，またそれによって最

†2　“`-Yclosure-elim`”, “`-Yconst-opt`”, “`-Ydead-code`”, “`-Yinline`”, “`-Yinline-handlers`”

表 1 *fib*(30)

領域特化言語	時間 (s)
逐次	0.004
Scala Async	0.165
shift/reset	0.598
Ticket	1.734

適化が阻害されること，継続渡しのためにクロージャを大量に生成すること，さらに継続を 2 回切り取るため *shift/reset* と比較しても単純計算で 2 倍のクロージャが生成されることなどが原因として考えられる．パフォーマンスの改善は並列化において重要であり，今後の課題である．

5 結論

本稿では，ticket 演算子を導入し，Java 8 のフューチャーオブジェクトのようなフォーク・ジョイン型の並列化プリミティブに対して，埋め込み領域特化言語の構文的な制約を軽減できることを示した．また，限定継続演算子の機能を取り込んだことにより，ノンブロッキング型の並列化プリミティブに対しても同様にできることを示した．

今後の課題としては，他の並列化プリミティブへの適用可能性の検証，並列計算における実用化に向けた実装の改善，Java を始めとする他の言語における実装を行う等が考えられる．

参考文献

[1] Asai, K. and Kiselyov, O.: Introduction to Programming with Shift and Reset, *ACM SIGPLAN Continuation Workshop 2011*, 2011.

[2] Danvy, O. and Filinski, A.: Abstracting control, *Proceedings of the 1990 ACM conference on LISP and functional programming*, ACM, 1990, pp. 151–160.

[3] EPFL and Typesafe, Inc.: Scala Async. `https://github.com/scala/async`, `http://docs.scala-lang.org/sips/pending/async.html`.

[4] Filinski, A.: Representing Monads, *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, New York, NY, USA, ACM, 1994, pp. 446–457.

[5] Intel Corporation: CilkPlus. `http://www.cilkplus.org/`.

[6] Joerg, C. F.: *The cilk system for parallel multithreaded computing*, PhD Thesis, Massachusetts Institute of Technology, 1996.

[7] Joyent, Inc: node.js. `http://nodejs.org/`.

[8] Rompf, T., Maier, I., and Odersky, M.: Implementing First-class Polymorphic Delimited Continuations by a Type-directed Selective CPS-transform, *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, New York, NY, USA, ACM, 2009, pp. 317–328.

[9] 浅井健一: shift/reset プログラミング入門, *ACM SIGPLAN Continuation Workshop 2011*, 2011.