より柔軟な非同期実行記述を可能にするライブラリのための新しい限定継続演算子に向けて

山口洋 [1], 千葉滋 [2]

[1] 東京大学情報理工学系研究科
hiroshi_yamaguchi@csg.ci.i.u-tokyo.ac.jp

[2] 東京大学情報理工学系研究科
chiba@acm.org

概 要 本研究では、より柔軟な非同期並列実行記述を可能にするライブラリのために新しい限定継続演算子の設計を試みる。具体的には、継続を切り取る機能に加えて、引数に依存するプリフィックスとそれ以外の 2 段構成に加工を行うようなものである。このような演算子を考える背景としては、非同期並列実行をライブラリレベルで簡単に記述したいというものがあり、その記述方法は、一旦代入を要するフューチャーオブジェクトのように専用のロジックでの記述を要求せず、式の中に非同期キーワードを用いてインラインで記述でき、逐次型からの修正を小さくする形式であることが望ましいと考えている。本稿では、これをフューチャーオブジェクトや shift/reset による直接形式などといった、従来手法により実装しようとする場合の問題点を議論するとともに、ライブラリレベルで実現することによる利点についても考察することで、新しい演算子の必要性を明らかにする。

1 はじめに

本研究では新しい限定継続演算子の設計を試みる。その目的は非同期並列実行をより柔軟に記述できるような EDSL (Embedded Domain Specific Language) を作ることにある。すなわち、EDSL という形で、ライブラリレベルで非同期並列実行を支援するようなプリミティブを提供できるように、より低レベルの言語機構として、新しい限定継続演算子を設計する。計算対象や、実行環境といった、非同期並列実行を記述する状況によって、計算ロジックを記述する上で最適な抽象度のプリミティブは様々であり、最高のパフォーマンスを上げる実装法も様々である。そのため、本研究では、言語として唯一の解となるプリミティブを直接提供するのではなく、EDSL の作成を支援する言語機構を提供することを目指している。

非同期並列実行で計算を行うプログラムを記述する上で考慮しなければならない点の 1 つとして並列化の粒度、すなわち並列化においてプログラムを分割する細かさがある。例えば、粗い粒度としてはメソッドを単位とすることが考えられるが、2 つの式の和を求める計算があって、それぞれの式を並列に計算したいときなど、メソッドの内部構造に注目してより細かく並列化したいことがある。一方で、並列化の粒度を式単位まで細かくすると、オーバーヘッドの原因となる。並列化の粒度は適切に選択しなければならないが、適切な粒度は対象となる問題やプログラムの構造だけではなく、問題サイズや使用可能なハードウェア資源など、動的なパラメータに依存することがよくあるため、静的に選択することは難しい。

我々はプログラムを EDSL を使って記述することで、EDSL ライブラリ側でプログラムの一部を分割し、分割した部分を非同期並列実行できるようにすることを目指す。つまり、EDSL ライブラリ側で粒度を変更でき、粒度を粗くする並列実行プリミティブや細かくするプリミティブなどが、それぞれライブラリとして実装できる。このようにすると、アプリケーションプログラマは、使用

したい EDSL ライブラリをプログラムから呼び出すことで、並列化の粒度の選択時に EDSL ライブラリの補助を受けられ、実行環境や動的な文脈に応じた選択が容易になる。

我々はこのような非同期並列実行 EDSL を、限定継続演算子を用いて実現する。限定継続演算子を用いることで、フューチャーオブジェクトに相当する非同期並列実行プリミティブが実現できることが知られている。しかしながら、限定継続演算子を用いて、非同期並列実行 EDSL を実際に設計したところ、従来の演算子では記述力が不十分であるという結論に至った。本稿では、その記述力の不足とその理由を述べるとともに、より柔軟な EDSL を実現できるような新しい限定継続演算子のプロトタイプを提案する。

本稿の構成は次のようになっている。まず 2 節で、フィボナッチ数という簡潔な例を題材に、3 種類の DSL によるプログラムを示し、従来の限定継続演算子の記述力の不足とその理由を指摘する。次に 3 節で、従来の限定継続演算子と新しい演算子の概要を述べる。最後に 4 節で、ライブラリレベルでなければならない理由、実際に限定継続演算子を使って非同期並列実行ライブラリを作る方法、新しい演算子以外の方法はないか、といった点について議論を深める。

なお、本稿では、サンプルコードの記述言語として主に Scala 2.10 を用いる。

2　非同期並列実行によるフィボナッチ数の計算

本節では n 番目のフィボナッチ数を計算するプログラムを例に、I 型、 II 型、III 型と名付けた、3 種類の DSL で記述した例を挙げ、非同期並列実行する DSL を構築する上での、限定継続演算子の表現力を検証する。フィボナッチ数を求めるプログラムは、実用的なプログラムと比較すると非常に単純であり、実用性などを論じるにはあまり向かないが、再帰的で不規則な構造を持っていて簡潔な例であり、表現力の検証には十分である。

まず、同期と非同期、2 種類の並列実行プリミティブについて本稿における定義を行う。同期並列実行プリミティブとは、フォークしたタスクの結果を受け取るときに、実際に値が返ってくるまで待つプリミティブのこととする。また、非同期並列実行プリミティブとは、フォークしたタスクの結果を受け取るときに、非同期並列実行コンテナを介するプリミティブのこととする。非同期並列実行コンテナは実際の値を取り出せない代わりに、結果を使って行う処理をイベントハンドラのように登録できる。なお、実際には、最終的に同期プリミティブを通じて非同期並列実行コンテナから値を取り出すことになる。そのため、多くの非同期並列実行コンテナには値を同期的に取り出す “裏口” が存在する。

それでは、DSL で記述したフィボナッチ数を計算するプログラムを示していく。まず、参考として、逐次実行のみで計算するフィボナッチ数のプログラムを図 1 に示す。今回用いるアルゴリズムは、数あるフィボナッチ数の計算法の中でも愚直に漸化式を再帰的に計算するものである。このプログラムの意味は、n が 2 未満のときは n を返し、それ以外のときは再帰的に実行して、$n-1$ 番目、$n-2$ 番目のフィボナッチ数を求め、それらの和を返す、というものである。以下では、このプログラム中の、$n-1$ 番目、$n-2$ 番目のフィボナッチ数を求める部分、を非同期並列実行させることによって並列化したプログラムを記述する。すなわち、

1. `val xa = fib(n - 1)`: タスクをフォークする
2. `val ya = fib(n - 2)`: タスクをフォークする
3. `val x = await(xa)`: 結果を待つ
4. `val y = await(ya)`: 結果を待つ
5. `x + y`: $x + y$ を計算する

といったプログラムで表せるような実行の流れを持つプログラムである。

```
1  def fib(n: Int): Int =
2    if (n < 2)
3      n
4    else
5      fib(n - 1) + fib(n - 2)
```

図 1. 逐次実行のみで計算するフィボナッチ数のプログラム

```
1  def fib(n: Int): Async[Int] =
2    async {
3      if (n < 2)
4        n
5      else
6        inline_await{ fib(n - 1) } + inline_await{ fib(n - 2) }
7    }
```

図 2. I 型の DSL を用いた非同期並列実行されるフィボナッチ数のプログラム

まず、I 型の DSL で記述したものが、図 2 に示すプログラムである。この DSL は、今回扱う DSL の中では逐次実行のみで計算するプログラムとの対応関係を最もよく残したものである。この DSL では次の 3 種類のプリミティブを提供する。まず、`async` は指定されたブロックを非同期並列実行をすることを示すキーワードである。次に `Async` 型は `async` によって非同期並列実行による結果であることを示す。最後に `inline_await` は `Async` 型のオブジェクトから値を取り出すためのキーワードである。[1][2]

最初の逐次実行されるプログラムと比較すると、この I 型の DSL を使ったプログラムには、

1. 6 行目の `fib` の呼び出しが `inline_await` ブロックで囲まれている、
2. 2–6 行目の関数本体の定義が `async` ブロックで囲まれている、
3. 1 行目の返り値の型が `Int` ではなく `Async[Int]` である

といった 3 点の違いがある。なお、最終的には `fib` を同期プリミティブを用いて呼び出す必要があるが、話を簡単にするためその部分のことは無視する。

次に II 型の DSL で記述したプログラムが図 3 に示すプログラムである。I 型の `inline_await` の代わりに `await` が提供され、I 型とよく似ているが、実は全く異なる DSL である。詳しくは後述するが、`await` は `inline_await` とは異なり、非同期並列実行するブロックを直接渡すのではなく、代入を介して間接的に渡さなければならないという呼び出し規約を持っている。図のプログラムは、`xa`、`ya` への代入時に、内部に `async` を含んでいる `fib(n-1)`、`fib(n-2)` をそれぞれ非同期並列実行し、`await(xa)`、`await(ya)` でそれぞれ計算結果を待つ。この II 型に類する DSL を提供する非同期並列実行ライブラリあるいはフレームワークは多く、Scala Async [4]、.NET の `async/await` [2]、Cilk2 [12] などがある。同期 DSL という点で少し離れるがフューチャーオブジェクトを用いたプログラムもこれに類似した外観を持つ。

最初の逐次実行されるプログラムと比較すると、この II 型の DSL を使ったプログラムには、

1. 8 行目の `xa`、`ya` の呼び出しが `await` ブロックで囲まれている、
2. 2–8 行目の関数本体の定義が `async` ブロックで囲まれている、
3. 1 行目の返り値の型が `Int` でなく `Async[Int]` である

といった 3 点の違いがある。

[1] Scala 2.10 では `Async` に相当するものは `scala.concurrent.Future` として提供されている。しかしながら、古典的な同期点でブロックするフューチャーオブジェクトと紛らわしいためこの名前を用いる。

[2] 本稿におけるフューチャーオブジェクトは、MultiLisp の `future` のような評価規則を拡張するタイプではなく、Java、Racket 等で採用されているようなオブジェクトでラッピングすることで通常の評価規則の中に埋め込んだ、`get` あるいは `touch` といった明示的な値の取り出し操作を必要とするタイプのことである。

```
1  def fib(n: Int): Async[Int] =
2    async {
3      if (n < 2)
4        n
5      else {
6        val xa = fib(n - 1)
7        val ya = fib(n - 2)
8        await(xa) + await(ya)
9      }
10   }
```

図 3. II 型の DSL を用いた非同期並列実行されるフィボナッチ数のプログラム

```
1  def fib(n: Int): Async[Int] =
2    if (n < 2)
3      Async { n }
4    else {
5      val xa = fib(n - 1)
6      val ya = fib(n - 2)
7      xa.andThen { x =>
8        ya.andThen { y =>
9          Async { x + y }
10       }
11     }
12   }
```

図 4. III 型の DSL を用いた非同期並列実行されるフィボナッチ数のプログラム

前述したように I 型と II 型の違いは DSL によって要請される非同期並列実行の呼び出し規約にある。I 型では図 2 のように非同期並列実行が式中にインラインで記述できるのに対し、II 型では図 3 のように一旦代入するという手順があり、専用のロジックに従った記述を要する。軽微な違いに見えるかもしれないが、この 2 つの DSL の間には互換性がなく、本質的な違いがある。II 型の DSL が多くのライブラリ・フレームワークで採用されているのに対し、I 型の DSL は通常の言語で実現することが困難である。

まず、I 型の DSL を作ることができない理由を述べる前に、II 型の DSL を実装する方法を示す。II 型の DSL であってもすべての言語で EDSL として実装できるわけではない。例えば、 Java、C++ のようなよく使われる言語は機能が不十分である。なぜなら、II 型の DSL では今の実行状態、すなわち、ローカル変数の値、プログラム中の実行中の場所などを保存してイベントハンドラとして非同期並列実行コンテナに登録する必要があるが、そのような機能は備わっていないからである。詳細は 4.2 節で解説するが、このような言語では III 型の DSL がよく用いられる。図 4 は III 型の DSL によるフィボナッチ数を求めるプログラムである。`Async` で非同期並列実行コンテナを作り、図 3 の 8 行目のような同期点で明示的に計算を分割し、ブロックとして `andThen` メソッドに与え、合成する。`andThen` メソッドは、非同期実行コンテナの計算結果を待ち、その結果を用いて与えられたブロックを実行する。III 型の DSL は II 型の DSL と比較すると極めて煩雑である。

II 型の DSL を実装できるのは継続演算子を持つような言語である。継続演算子を用いると、明示的にプログラムを分割することなく、適切なところでプログラムを分割することができるためである。詳しくは 4.2 節で述べるが、実際、継続演算子の 1 種である *shift/reset*[8, 7, 15] の場合、`await` を *shift*、`async` を *reset* で実現できる。

継続演算子を使えば II 型の DSL は EDSL として実現できるものの、I 型の DSL は実現できない。これは、逐次型言語の評価戦略のためである。I 型、II 型のような DSL が存在したと仮定して、逐次型言語ではどのように評価されるか、実際に示すことにする。ここでは、簡単のために継続演算子はないものとし、プログラムは単純に left-to-right の値呼びで評価されることにする。この評価戦略の下で、`f()+g()` のような式は、

1. `val fv = f()`
2. `val gv = g()`
3. `fv + gv`

と書き下したものと同じように評価される。すなわち、第 1 引数、第 2 引数、本体の順である。ここで、図 3 について見てみると、6–8 行目は

1. `val xa = fib(n - 1)`: タスクをフォークする
2. `val ya = fib(n - 2)`: タスクをフォークする
3. `val x = await(xa)`: 結果を待つ
4. `val y = await(ya)`: 結果を待つ
5. `x + y`

のようになり、`fib(n - 1)` と `fib(n - 2)` が正しく並列に計算される。しかしながら、図 2 の 6 行目の場合は、

1. `val xa = fib(n - 1)`: タスクをフォークする
2. `val x = inline_await(xa)`: 結果を待つ
3. `val ya = fib(n - 2)`: タスクをフォークする
4. `val y = inline_await(ya)`: 結果を待つ
5. `x + y`

のように、第 2 引数を評価するタスクがフォークされる前に第 1 引数の結果を待ってしまい、並列実行になっていない。正しく並列に評価されるためには、第 1 引数の結果を待つことなく、第 2 引数の評価を始めなければならない。すなわち、left-to-right ではなく、すべての引数を並列に評価しなければならないが、これを逐次実行に基盤を置く言語上のライブラリとして実現することは難しい。これは、従来の限定継続演算子がある場合も 3.1 節で説明するように状況が大きく変わることはない。

3 新しい限定継続演算子 pshift

このように、既存の言語機構で I 型のようなインライン形式を実現するのは困難である。本節ではこれを実現するような言語機構を導入することを試みる。前節の議論を踏まえ、本研究では I 型の DSL を実現する言語機構として限定継続演算子に注目し、それを改良した新たな演算子を導入することを提案する。

3.1 従来の演算子

先に、従来の限定継続演算子について簡単に説明する。まず、継続とは、プログラムの実行中の特定の時点において、これから実行すべき残りの計算のことを指す概念である。また、継続演算子とは、継続を切り取り、無名関数、クロージャ、オブジェクトなどの形でプログラムに渡し、プログラム中で明示的に扱えるようにする、リフレクション系の演算子のことである。古典的には、プログラムの終了まで継続全体を切り取る継続演算子が長く用いられてきたが、*goto* と同等であってモジュール化などの観点などから好ましくないため、近年においては切り取る範囲に制限を加えた演算子が用いられる。そのような演算子は、切り取る範囲を制限する演算子と制限された範囲内で継続を切り取る演算子の組で与えられることが多く、これらの組を総称して限定継続演算子という。

限定継続演算子の代表格としては *shift/reset*[8, 7, 15] や *control/prompt*[9] などがある。おおよそ、*reset* や *prompt* が範囲を制限する演算子で *shift* や *control* が継続切り取る演算子である。なお、*shift* は単純に切り取るだけでなく、同時に、切り取った継続の内部に存在する演算子が切り取る範囲を、自分が切り取った継続内に制限する機能を持つ点が特徴的である。本稿では、従来の限定継続演算子として主に *shift/reset* を扱う。Scala においても、バージョン 2.10 現在、CPS プラグイン [14] と呼ばれるコンパイラプラグインが標準で提供され、*shift/reset* をプログラム中で使用することができる。

限定継続を扱うときには型を考えることも重要である。関数においては、通常の返り値型に加えて、切り取る継続の返り値型 γ と切り取った継続を使った後の結果型 δ が必要になる。これを、Scala 2.10 では、$@cpsParam\,[\gamma, \delta]$ という型アノテーションで表現し、関数全体では、

$$\alpha \Rightarrow \beta @cpsParam\,[\gamma, \delta] \tag{1}$$

のように書く。また、*shift/reset* チュートリアル資料 [7, 15] などで使われている表記法に従うと、

$$\alpha/\gamma \to \beta/\delta \tag{2}$$

のようになる。また、

$$@cps\,[\varepsilon] = @cpsParam\,[\varepsilon, \varepsilon] \tag{3}$$

という略記もよく使われる。

3.2 新しい演算子

I 型の DSL を実現する上で限定継続演算子 *shift/reset* に足りないものを検討する。I 型の DSL を実現する上で障害となったのは引数の評価順序であり、この順序を変える手段が必要なのであった。プログラムの一部を切り取って、実行順序を変えるというのは、限定継続演算子とよく似ている。しかしながら、*shift/reset* では関数本体と未評価の引数を必ずひとまとめにして切り取ってしまうため、それらを切り離すことはできない。また、*shift/reset* で渡される継続の実態は無名関数かそれに類するものであるため、引数を受け取らずに実行するといった選択肢も存在しない。

これを踏まえ、本研究では、*shift/reset* を引数を受け取ることなく実行を進められるように改良し、切り取った継続を加工して渡す新しい演算子 *pshift/reset* を提案する。この演算子は、継続を切り取る機能に加えて、引数に依存しないプリフィックス部分とそれ以外に分けるような加工を行う。ここでいう引数に依存しないプリフィックス部分とは、継続がステートメントの列だと考えたときに、先頭から引数に依存するステートメントの直前までの部分のことを指す。ただし、継続の引数に依存しないプリフィックスを考える上では、Java の `return` 文のように値を返すだけのものは依存しているとみなさないことにする。直感的には、渡された関数からこの引数に依存しないプリフィックス部分を外に出してサンクを作り、そのサンクを評価すると、継続の本質的な部分が濃縮された関数が手に入る。言い換えると、1 段目のサンクと 2 段目の濃縮された継続の 2 段構成で継続を渡すということになる。

この演算子は、自由度が 1 段目のサンクの評価タイミングを指定できる分だけ高くなっているが、従来の演算子も同様であるが、切り取った継続を実際にどう使うかはプログラム上で明示的に指示できる。そのため、演算子の役目は、DSL で書かれたプログラムを非同期並列実行可能なように加工するだけであり、非同期並列実行機能そのものは他の言語機構が提供したものに依存している。例えば、この演算子のようにサンクを作るのではなく、演算子が使用されたタイミングで非同期に部分評価を行い、濃縮された継続を渡す、といった演算子も考えられるが、そういった言語機構側で非同期並列実行を行うようなことはしないということである。使い方を明示的に指定できることにより、また、2 段化には、サンクの評価をスキップする、非同期並列化せず元の順序のまま実行するといったことを可能にするといった利点もある。

新しい演算子 *pshift/reset* の意味をもう少し詳細に示す。説明には、ラムダ式に逐次実行の ; と変数への束縛の $\leftarrow$ を加えたものを用いる。

$$reset\,\{\lambda\_.\,\{\ldots;x \leftarrow pshift(M);A;\ldots;F(x);B;\ldots\}\} \tag{4}$$

のようなプログラムがあったときに、継続として M に与えられるのは、

$$\lambda\_.\,\{A;\ldots;\lambda x.\,\{F(x);B;\ldots\}\} \tag{5}$$

である。継続の引数 x に依存しないプリフィックス $A;\ldots$ からなるサンクが 1 段目にあり、その上に 2 段目として引数 x に依存する $F(x);B;\ldots$ からなるラムダ項が載っている、2 段構成の形である。継続の引数 x に無関係な部分がサンクとしてラムダ項の外に置かれているため、計算結果を待たず評価することが可能であり、継続の引数が必要な部分は、サンクを評価することで引数 x を取るラムダ項として手に入るようになっている。従来の演算子 *shift* では、*pshift* のときのような 2 段構成にならず、単純に

$$\lambda x.\,\{A;\ldots;F(x);B;\ldots\} \tag{6}$$

となり、通常の評価規則の下で、x に無関係な $A;\ldots$ の部分を評価するといったことはできない。

Reset で囲まれた内側に複数の演算子が存在した場合は次のようにする。2 段構成になっているため、2 個目の演算子が 1 段目中にあった場合と 2 段目中にあった場合とで、違いを考慮しなくてはならない。まず、2 段目にあった場合を考える。新しい演算子 *pshift* も、*shift* と同様に、切り取ると同時に、内側の演算子が切り取る範囲を外側の演算子が切り取った継続内に制限する、ということにする。これにより、単独で存在した場合と全く同じように扱うことができる。1 段目中にあった場合は、1 個目の演算子によって 2 段構成に書き換えられたと考えて、演算子の適用を行うことにする。すなわち、

$$reset\,\{\lambda\_.\,\{\ldots;x \leftarrow pshift(M);y \leftarrow pshift(N);A;\ldots;F(x);B;\ldots;G(y);C;\ldots\}\} \tag{7}$$

において、継続として N に渡されるのは、

$$\lambda\_.\,\{A;\ldots;\lambda y.\lambda x.\,\{F(x);B;\ldots;G(y);C;\ldots\}\} \tag{8}$$

ということになる。

式を見るとわかるように、このプロトタイプおいては、1 個目の演算子によって切り取られて 2 段目になった部分は、たとえ 2 個目の演算子において、引数に関係ない部分だったとしてもプリフィックスとして取り出されることはないということである。つまり、

$$x \leftarrow pshift(M);y \leftarrow pshift(N);G(y);F(x) \tag{9}$$

のように変数への束縛と最初の使用の位置関係が入れ子状になっているときには問題ないが、

$$x \leftarrow pshift(M);y \leftarrow pshift(N);F(x);G(y) \tag{10}$$

のように千鳥状になっているときには非同期で実行できる範囲が期待よりも少なくなってしまう可能性がある。この制約は、n-項演算子の引数の評価順序を並列にするという今回の目的においては

$$x \leftarrow pshift(M);y \leftarrow pshift(N);F(x,y) \tag{11}$$

のように使う位置がほぼ同じであるためほとんど影響しないが、今後他の用途に使うことを考えるときには注意しなければならないと思われる。

また、ラムダ抽象の内側に *pshift* があり、*reset* が外側、すなわち関数適用側にあるときは次のようにする。提案する演算子では、

1. ラムダ抽象の内側が継続の引数に依存しないプリフィックスのみからなるとき、関数適用した先にも *pshift* を置いたと考えて演算子のサンクに追加する、
2. ラムダ抽象の内側に継続の引数に依存しないプリフィックス以外を含むとき、それ以上演算子のサンクに追加しない、

と定める。この演算子において、 *pshift* によって作られる 1 段目のサンクには、ラムダ抽象の外にある部分も含まれうる一方、その条件は限られており、通常は DSL に直接関係する限られた範囲のプログラムだけがその対象となるようになっている。そのため、通常の使い方においては、1 段目のサンクに入るかどうかを考慮しなければならない範囲が際限なく大きくなるといったことは心配ない。

これらをまとめると、*pshift* を使ったフィボナッチ数を求めるプログラムは図 5 のように書ける。このプログラムでは、インラインの非同期並列化プリミティブとして `inline_await1` と `inline_await2` を使い、それぞれの内部で *pshift* を使っている。この例では引数が 2 個であるが、n 個の引数に一般化した場合にも非同期並列化プリミティブは 2 種類となる。一番右側の引数だけは `inline_await2` のように継続の型に `Async` が入らないプリミティブを使い、残りの引数に対しては `inline_await1` のように継続の型が `Unit=>Async[...=>...]` の形になるものを使う。この 2 種類に分かれたプリミティブは I 型の DSL と異なっており、*pshift* の不完全な点である。

それでは `inline_await1` と `inline_await2` の解釈を説明する。まず、`inline_await1` では、*pshift* により切り取った継続を分解する。このとき、1 段目が `inline_await2{fib(n-2)}` の部分、2 段目が`+`の部分になり、結果として `Unit=>Int=>Int` 型のクロージャとなる。次に、1 段目に *pshift* を含む `inline_await2` があるので、同じように継続を切り取って分解すると 1 段目が空、2 段目が`+`の部分となり、`inline_await1` のときより引数が 1 個増えて `Unit=>Int=>Int=>Int` 型のクロージャとなる。`inline_await2` の *pshift* の内部にこのクロージャを渡すと、まず渡されたブロックを非同期並列実行するプリミティブ `Async` を用いて、`fib(n-2)` を非同期に計算し、コンテナを合成するプリミティブ `andThen` を用いて継続を適用する。すると、`Async[Int=>Int]` 型のクロージャが得られ、`inline_await1` の方に返される。ここで、`inline_await1` の受け取る継続の型は本来、`Unit=>Int=>Int` 型であったが、`inline_await2` において異なる型を返す書き換えを行ったため `Unit=>Async[Int=>Int]` になった。`inline_await1` もまた `inline_await2` とおおよそ同等のことをしている。しかしながら、`inline_await2` とは継続の型のパターンが異なるので、`andThen` を使った合成方法などに多少の違いがある。なお、この継続の型のパターンと合成方法の違いからこの 2 つのプリミティブを統合することは困難である。

図 5 のプログラムの実行の流れはほぼ図 2 の説明で述べた通りである。ただ、細かいところでは異なる点もあり、例えば、`inline_await2{fib(n-2)}` から `inline_await1{fib(n-1)}` への結果の受け渡しで、コンテナに包まれているものが、`fib(n-2)` の計算結果ではなく、`fib(n-1)` の計算結果を使うクロージャ、すなわちその呼出時点における継続である点が 1 つである。

このように、複雑な仕組みではあるが、おおよそ I 型の DSL の主要部は実現できたと考えられる。しかしながら、前述したように `inline_await` が 2 種類に分かれる点など不完全な点が残っている。

3.3 未解決の課題

本稿で提案する演算子 *pshift* は開発中であり、いくつか未解決の課題が残っている。まず、1 つ目は直前に述べた図 2 の `inline_await` を実現できていない点である。言い換えると、`inline_await2` を `inline_await1` に統合できていないこと、すなわち、継続として `Unit=>Int=>Int=>Int` 型ではなく、`Unit=>Async[Int=>Int=>Int]` 型のクロージャが渡されるようにする方法が見つかっていないことである。よく似た II 型の DSL では、4.2 節に示すように、型書き換えを利用することなく、統合されているのとは対照的である。これが *pshift* の本質的な問題に起因するのかは不明である。

2 つ目は、前に述べた *pshift* の入れ子型のフローへの制限である。本研究では今後の展開として、並列ワークフローエンジンやデータフロー指向の EDSL への応用も考えているが、この制限が大きな障害の 1 つとなっている。また、I 型の DSL は II 型の上位互換である、すなわち、I 型の DSL のプリミティブを II 型の DSL のように並列化を意識して使った場合には分岐・合流をこと細かに指

```
def fib(n: Int): Async[Int] =
  if (n < 2)
    Async { n }
  else {
    reset {
      inline_await1{ fib(n - 1) } + inline_await2{ fib(n - 2) }
    }
  }
def inline_await1(t: => Int) = {
  pshift { (k: Unit=>Async[Int=>Int]) =>
    val a = Async { t() }
    val ka = k()
    a.andThen { v =>
      ka.andThen { kv =>
        kv(v)
      }
    }
  }
def inline_await2(t: => Int) = {
  pshift { (k: Unit=>Int=>Int=>Int) =>
    val a = Async { t() }
    val kv = k()
    a.andThen(kv)
  }
```

図 5. 限定継続演算子プロトタイプを用いたフィボナッチ数の計算

定できる、という性質を満たしていてほしいというのは自然な需要であるが、やはりこの制限により、少なくとも単純な方法では、期待した通りに動くような DSL を実装できない。

3 つ目は、おそらく 2 つ目に関係すると思われるが、*pshift* は *shift* の上位互換であるか。という疑問である。これが示されれば、新しい限定継続演算子で *shift* を置き換えることが可能である。ただし、おそらく現行の提案演算子はこの性質を持たないと予想しており、そのため改良を試みている。

これらが今後、提案演算子が解決すべき課題である。

4 議論

以下、ここまでの議論で幾つか詳細を略した点についてもう少し議論する。

4.1 非同期 DSL をライブラリレベルで実現する利点

非同期 DSL をライブラリレベルで実現する利点として 2 つの例を紹介する。

1 つ目は、DSL 開発者あるいは DSL 利用者の開発したカスタムスケジューラを利用できるということがある。スケジューリングの戦略は非同期並列実行の速度を決めるだけでなく、メモリ使用量などの必要リソースにも大きな影響を与える重要な要素である。そればかりか、場合によってはリソース不足による異常終了の原因にもなりうる。スケジューリング戦略によっては計算できなくなる例としては、繰り返し、例として使っているフィボナッチ数がその 1 つである。フィボナッチ数の計算においては、良い順序でタスクを処理すれば、同時にスケジューリング対象となるタスクの数は引数のオーダーになる一方で、良くない順序では、同時にスケジューリング対象となるタスクの数は返り値のオーダーになり、タスクの保管に必要なコストが指数的に爆発してしまう。このように、ユーザー側で変更可能なスケジューラがあると便利であり、これはライブラリレベルで実装されていると都合が良い。

もう 1 つは、DSL 開発者あるいは DSL 利用者が適切な粒度を選ぶ最適化を施しやすくなるということがある。一般に、非同期化は大きなオーバヘッドを伴うものであり、適切な粒度を選択しなければ逆に遅くなる原因となってしまう。しかしながら、すべてのプログラム、実行環境に対応で

```
trait Monad[M[+ _]] {
  def unit[A](x: A): M[A]
  def bind[A, B](m: M[A], f: A => M[B]): M[B]
}
```

図 6. モナドの定義

```
def fib(n: Int): Async[Int] =
  if (n < 2)
    Async { n }
  else {
    val xa = fib(n - 1)
    val ya = fib(n - 2)
    xa.flatMap { x =>
      ya.flatMap { y =>
        Async { x + y }
      }
    }
  }
```

図 7. 図 4 の `flatMap` による記述

きるような選択戦略を見つけることは困難であり、ライブラリレベルで実装して柔軟性を保っておいた方が便利である。

4.2 Scala における II 型の DSL の実装

一般に、II 型の DSL は III 型の DSL の上に、*shift/reset* を使った直接形式と呼ばれるものにより構築することができる。III 型の DSL に関する詳しい説明と、それを使って II 型の DSL を構築する方法について説明する。

まず、 III 型の DSL は、図 4 のように、同期点より下を明示的に無名関数として切り離して、メソッドチェーンでつなぐようなスタイルをとる。代表格としては、Node.js[3] があり、継続演算子を持たない言語で広く使われるようになってきている。

この III 型の DSL は近年モナド [13] として知られるようになったデザインパターンに基づいてる。モナドとは図 6 に示すような *unit* と *bind* の 2 つのメソッドを持つコンテナであり、その中でモナド則と呼ばれるいくつかの良い性質を満たすものである。Scala では *unit* と *bind* は、純粋な値を 1 つだけ引数に取るコンストラクタと `flatMap` におおよそ相当し、Haskell では `return` と `>>=` におおよそ相当する。

ここで、*unit* とは純粋な値をコンテナにラップするメソッドであり、*bind* とは純粋な値を引数に取るクロージャをコンテナに適用するメソッドである。この 2 つのメソッドにより、モナドはコンテナから値を取り出す操作を隠蔽したまま、クロージャを適用していくことができるようになっている。また、返り値が純粋な値でなくても良いので、他のコンテナとの合流も扱うこともできる。非同期並列実行においては、イベントハンドラのように次にすべき処理を登録する操作を *bind* と考えると、非同期並列実行の実際には値を取り出さないというモデルがうまく表現される。III 型の DSL をモナドとして捉え直すと、図 4 における `andThen` は *bind* あるいは、`flatMap` のことであり、書き下すと図 7 のようになる。

また、一般にモナドと呼ばれるコンテナは、*shift/reset* と組み合わせることで、直接形式あるいは reflective 形式 [10] などと呼ばれる形式のコンテナに変換できることが知られている。直接形式は、擬似的にコンテナから値を取り出す *reflect* と、擬似的に取り出した値を使って得られた値をコンテナに戻す *reify* の 2 つのメソッドからなる。これにより、コンテナで値が包まれていることを意識させにくい形式で記述できるようになる。

```
1   def reflect[A,B](m: M[A]): A@cps[M[B]] =
2     shift { (k: A => M[B]) =>
3       bind(m, k)
4     }
5   def reify[B](x: => B@cps[M[B]]): M[B] =
6     reset{ unit(x) }
```

図 8. Scala における直接形式のモナドの定義

```
1   def reflect[A](m: M[A]): A@cps[M[Any]] =
2     shift { (k: A => M[Any]) =>
3       bind(m, k)
4     }
5   def reify[B](x: => B@cps[M[Any]]): M[B] =
6     reset(unit(x)).asInstanceOf[M[B]]
7   def thunkingReify[B](x: => B@cps[M[Any]]): M[B] =
8     bind(unit(), ((_: Unit) => reify(x)))
```

図 9. Scala 上で非同期並列実行を実現するために修正したもの

直接形式は図 8 のように実装できる。これについてもう少し詳しく説明する。まず、*reflect* は擬似的に値を取り出すことを表現するために内部で *shift* を用いている。*Shift* により限定継続を切り出し、それを *bind* でコンテナに適用することにより、前節のモナドの計算に翻訳している。一方、*reify* は *reset* を用いて擬似的に値を取り出したことによる影響範囲を制限すると同時に、*unit* で普通のコンテナに値を戻している。

II 型の DSL の実装はこの直接形式への変換に相当し、`await` が *Async* モナドにおける *reflect*、`async` が *Async* モナドにおける *reify* である。

ただし Scala 上で非同期並列実行 DSL として用いる場合には、2 つの修正を加える必要がある。1 つは Scala の型推論方法に起因するものであり、図 8 中の `reflect[A,B]` において、型パラメータ `B` の型を適切に推論できないという問題である。1 つの解決策は、Rompf の論文 [14] に示されるなど、Scala で広く行われている方法で、`reflect` の使用の度に毎回型注釈を与えるものである。しかしながら、ここで与えなければならない型パラメータ `B` の型は最終結果のみに依存するものであり、個々の `reflect` の使用には全く依存しない冗長度の高い型注釈である。ライブラリの利便性を大きく損なってしまうだけの存在である。型注釈を使用しない解決策は、Fillinski の論文 [10] に SML/NJ[6] 向けのものが示されている。最上位型 $\top$、Scala でいうところの `Any` と、キャストを用いる。なお、キャストを使うため局所的に型安全性を損なってしまうように見えるかもしれないが、無関係な *shift/reset* が挿入されていない限りは安全である。これにより、型注釈を除去できる。

もう 1 つは、非同期並列実行コンテナ固有の性質に起因するものであり、予期しない逐次実行が発生してしまう問題である。本来 *unit* は値を受け取ってコンテナで包むものであり、値を計算する過程には注目していないが、非同期並列実行する上で欲しいものは値ではなく未評価のサンクである。この差異のため、*reify* のときに一部の計算が非同期並列実行の文脈から漏れてしまう可能性がある。これを防ぐためには、サンクを構築して非同期並列実行コンテナの文脈で実行されるように新しく *thunkingReify* を定義すればよい。

これら 2 つの改善をまとめると、図 9 のようになる。最終的に III 型の DSL を持つ非同期並列実行コンテナライブラリ `Async` を用いて、図 10 のようにして II 型の DSL を実装することができる。

4.3 他のアプローチによるインライン化を許す DSL の実装

多項演算子による数式表現において、インライン形式で非同期並列実行を表現するために、本研究では非同期並列実行により評価してほしい引数にアノテーションを付けるという視点から新しい限定継続演算子を導入することを提案した。他の方法としては、逆に演算子の方にアノテーション

```
object AsyncMonad extends Monad[Async] {
  def unit[A](x: A) = Async(x)
  def bind[A, B](m: Async[A], f: A => Async[B]) = m.flatMap(f)

  def async[T](body: => T@cps[Async[Any]]): Async[T] = {
    thunkingReify(body)
  }
  def await[T](a: Async[T]): T@cps[Async[Any]] = {
    reflect(a)
  }
}
```

図 10. Async/await の定義

を付けることによってインラインで表現する方法も考えられる。

実際にこれは存在しており、例えば、MultiLisp[11] などにおける *pcall* がある。Lisp で記述すると、

```
(pcall + (fib (- n 1)) (fib (- n 2)))
```

のように書くことによって部分的に引数が並列に評価されるようになる。同様に、Scala においても scalaz[5] というライブラリにおける ApplicativeBuilder を使った方法がある。やや不格好でインライン形式に含めてよいかは怪しいが、

```
(Async(x) |@| Async(y)) {_ + _}
```

のように書くことによって直接非同期並列実行コンテナを受け取るようにでき、これを使って演算子を並列版に変換するメソッドを作ることができる。

このアプローチに基づいた DSL の方がシンプルでわかりやすい場合も多いと思う。一方で短所もあって、この DSL ではフォークする時点の調整が少々煩雑になる。どちらのアプローチも一長一短である。

5 まとめと今後の課題

本研究では、フィボナッチ数を求めるプログラムを例に 3 種類の非同期並列実行ライブラリの DSL を考え、従来の言語機構による表現力の限界を明らかにするとともに、その解決のため新しい限定継続演算子の導入を提案した。

今後の課題としては、提案した新しい限定継続演算子は未完成であるため、その改善を行うとともに、実際に非同期並列実行ライブリをライブラリを開発して、パフォーマンスなど実用性に近い観点からも考察を深めることが必要だと考えている。

参考文献

[1] The Haskell 98 language report. http://www.haskell.org/onlinereport/.

[2] .NET async/await. http://msdn.microsoft.com/ja-jp/library/hh156528.aspx, http://msdn.microsoft.com/ja-jp/library/hh156528.aspx.

[3] node.js. http://nodejs.org/.

[4] Scala Async. https://github.com/scala/async, http://docs.scala-lang.org/sips/pending/async.html.

[5] scalaz. https://github.com/scalaz/scalaz, http://typelevel.org/projects/scalaz/.

[6] SML/NJ. http://www.smlnj.org/.

[7] Kenichi Asai and Oleg Kiselyov. Introduction to programming with shift and reset. In ACM SIGPLAN Continuation Workshop 2011, 2011.

[8] Olivier Danvy and Andrzej Filinski. Abstracting control. In Proceedings of the 1990 ACM conference on LISP and functional programming, pp. 151–160. ACM, 1990.

[9] Mattias Felleisen. The theory and practice of first-class prompts. In Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 180–190. ACM, 1988.

[10] Andrzej Filinski. Representing monads. In Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '94, pp. 446–457, New York, NY, USA, 1994. ACM.

[11] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. ACM Trans. Program. Lang. Syst., Vol. 7, No. 4, pp. 501–538, October 1985.

[12] Christopher F. Joerg. The cilk system for parallel multithreaded computing. PhD thesis, Massachusetts Institute of Technology, 1996.

[13] Eugenio Moggi. An abstract view of programming languages. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1990.

[14] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming, ICFP '09, pp. 317–328, New York, NY, USA, 2009. ACM.

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