Beautiful Error Handling

田中 英行 *

今日に於いても、プログラマにとってエラーハンドリングは頭の痛い問題であ る。

正しいエラーハンドリングを行うことは、堅牢なソフトウェアを構築する上で 大変重要な問題であるが、適切な抽象化が行われてきたとは言い難い状況にあ る。

現在広く用いられている手法としては、古くからのエラーコードによるものと、 例外によるものがある。エラーコードによる手法には大きく2つの問題がある。 エラー処理のためのコードがコードのあらゆる部分に組み込まれ、コードの見 通しが非常に悪くなることと、エラーを無視するのが非常に容易なので、予期 せぬエラー処理漏れを犯しやすいことである。一方で例外を用いる方法は、正 しいリソース管理とともに用いられる必要がある。メモリの管理をGCに任せた 場合でも、GCの管轄外のリソースは正しく処理されないといけない。これを怠 れば、多くの場合深刻なバグを引き起こすことになる。このような処理をもれ なく記述すると、往々にして例外捕捉のために深いネストが必要となる。例外 処理を正しく行うには、例外の正しい伝搬も必要であるが、これをきちんと行 うのはかなり面倒なため、いわゆる例外の握りつぶしが横行する結果となるケー スも少なくない。いずれの方法でも、エラーを正しく処理できているかどうか は簡単には推論できない。コードをじっくり観察する必要がある。

かくして現代のプログラマは、退屈で面倒な、容易に間違いを犯し得る、そし て何より「美しく」ないコードを神経をすり減らしながら書かなければならな い状況にあり、また、ともすればそれを良しとする風潮すら見受けられる。し かしながらそれは解決できる問題であり、単にプログラミングモデルからくる 制限に他ならないと考える。

本発表では、モナドによってエラー処理の抽象化を行うための手法を紹介する。 それとともにエラー処理に欠くことのできない、リソース管理の抽象化もあわ せて紹介する。モナドを用いて、例外処理のための雑多のコードを、いわゆる「モナドの床下配線」によって隠蔽する方法を、実際のHaskellにおける応用 例とともに見てゆく。それにより、エラーの通知側とそれを受け取って処理す る側が完全に分離され、用途にマッチしたエラーハンドリングを自由に記述で きるようになる。最終的に、正しいエラーハンドリングを美しく行えることを 示す。

発表内容

発表内容は次ページ以降のスライド資料に示す。

* 株式会社プリファードインフラストラクチャ 研究開発部門

質疑応答

質問（小川清さん） 最初職人芸でないということを何だか良いことのように言われたんですけど、結局職人芸をしなければいけないという発表ですか?

回答 職人が作ったものを凡人が使えるという発表です

質問（小川清さん） 職人が作ったものを凡人がつかえるのならば、その凡人はプログラマである必要はなく、職人なのでは

回答 使うと作るの間には大きな隔たりがあると考えます

質問（望月さん） エラーの伝搬についてはよくわかりましたが、今回の伝搬というのは、 基本的に同じレイヤーの伝搬ですよね。HTTP / network エラーにはレイヤ・レベルの変換に関するアプローチというのは何か治験はありますか?

回答 レイヤに対して多層なモナドの型クラスが考えられています。 その型クラスを定義しておけば、レイヤを変換するということができると いうようなものです

質問（望月さん） それは変換するためのフラグメントを書かなければいけない?

回答 そうですが各所について作る必要はないはずです

質問（サイボウズ・ラボ 三成さん） 例外とかどんなふうにつながるんですか? これと組み合わせられるんですか?

回答 エラーモナドから例外への変換と、 例外を投げるコードからエラーモナドへの相互変換コードを書けば 組み合わせられます

質問（三成さん） この枠組みをつかってれば大丈夫というのは?標準のIOを使うには?

回答 具体的には IO から MonadError に変換するの関数があるので、それを使うとこの枠組みで扱えるようになります

質問（川中さん） 非同期関数の返り値をコールバックで受け取るというスタイルがあると思 うんですけど、あれのエラーはどういう枠組みにすればいいか

回答 難しいので、非同期処理をラップしてスレッドにするとか。 非同期をそのまま使っているのは処理系の都合だと思うので、それから解消しないと難しいのではないでしょうか。非同期は本当に難しく、この枠組みでは何か拡張しなければ無理だと思い ます。ぜひアイデアを頂きたい

質問（酒井さん） Haskellはわからないが、エラー処理が汚くなるといのは、ここにエラー を処理したというのはJavaの例外で書いていれば難しいとおもうんですけ ど、どこに戻るかというのを綺麗に書くには?

回答 そもそもどこに戻るかというのを考えないで済む枠組みにする。下のブロッ クがしっかりしてれば上はただしいという積み上げで、戻るという問題に 関して考えなければ綺麗になる

質問（久野先生） できているのを見るとHaskellでは綺麗。1970年に例外処理が発展していて、いろんな言語に影響を与えている。Haskellで解決している問題が、他　の言語でも使うようになればより素晴らしいのでは。値にエラーを内包　するという考え方が進むと良いと思います。

回答　Haskellでは多くの人に広まりつつある考え方ですので、いずれは他の言　語でも実装例が出てくると思います

Beautiful Error Handling

田中英行 <tanakh@preferred.jp>
2012年夏のプログラミング・シンポジウム

自己紹介

- 田中英行 (@tanakh , http://tanakh.jp)
- (株)Preferred Infrastructure勤務のプログラマ
- Haskell 愛好家
  - BASIC(20年), C++(15年), Haskell(10年)
- 「すごいHaskellたのしく学ぼう!」(Learn You a Haskell for Great Good! の和訳)
  - 好評発売中!!

概要

- エラー処理に美しさを!
- エラー処理の抽象化
- Haskellでのアプローチ

エラー処理に美しさを!

背景

- エラー処理は醜くなりがち
- なんで汚くなるのか?
  - これまで適切な抽象化が行われて来なかったから
- なぜそういう状況になっているのか?
  - 大きな原因の一つはプログラミング言語の記述力の問題
  - fine-grained な処理の抽象化、リッチな型システム、etc...

エラー処理の例(1)

```
int foo(...)
{
  int fd = open(...);
  if (fd < 0) return -1; // <- error handling
  ...
}
```

エラー処理の例(2)

```
int foo(...)
{
  int fd = open(...);
  if (fd < 0) return -1; // <- error handling
  int fe = open(...);
  if (fe < 0) {          // <- error handling
    close(fd);           // <- error handling
    return -1;           // <- error handling
  }                      // <- error handling
}
```

エラー処理の例(3)

```
int foo(...)
{
  int fd = open(...);
  if (fd < 0) return -1; // <- error handling
  int fe = open(...);
  if (fe < 0) {          // <- error handling
    close(fd);           // <- error handling
    return -1;           // <- error handling
  }                      // <- error handling
  int ff = open(...);
  if (ff < 0) {          // <- error handling
    close(fe);           // <- error handling
    close(fd);           // <- error handling
    return -1;           // <- error handling
  }                      // <- error handling
  ...
}
```

一方…

そういう背景からか、

- エラー処理を Ugly に、愚直に書いてあるプログラムが良いプログラムである
- Ugly なエラー処理を、きちんともれなく書けるプログラマが良いプログラマである

という風潮も

なぜエラー処理に美しさが必要なのか?

- → プログラムに美しさが必要だから
- → なぜプログラムに美しさが必要なのか?
- → 多分、竹内先生が語って下さっているはず

所感

美しいプログラムは

- 完結で、理解しやすい
  - 理解したとおりに動作する
- 変更しやすい
  - 変更するときに考慮するべきことが少ない

つまり

- バグりにくい

(あくまでテクニカルに)

個人的な思惑

- プログラムを書くということを職人芸にしたくない
  - 現状、職人芸的なところは大きい
- 例えば、優れたプログラマだからといって
  - STLのアルゴリズムを実装できる必要はない
- なぜか?
  - ライブラリになっているから、それを使えばいい
- ではエラー処理は…?
  - 現状同じ構図ではない
  - 同じものをもたらしたい

何が必要なのか?

エラー処理の抽象化が必要である

- 抽象化されていれば
  - ライブラリの形として、再利用可能
  - 退屈なコードの繰り返しがなくなる
  - 誰でも簡単に正しいエラー処理が書ける

という理想の世界

エラー処理の抽象化

そもそもエラー処理とは

プログラムの実行中に発生するエラーに対して、正しく回復処理・あるいは報告を行い、何事もなかったかのように動き続ける堅牢なソフトウェアを書くための処理

- 報告して終了するだけで十分な場合もある
  - e.g. 単機能のコマンドラインプログラム

様々なエラー

- ハードウェア由来
  - 無線ネットワークが切れた * ディスクが壊れた
- 不正な引数
  - 存在しないファイル名でファイルを開いた
  - メールアドレス渡すところに名前渡した
- 呼び出し先の問題
  - タイムアウト
  - HTTP 503
- 単なるバグ
  - ぬるぽ, assertion failure, etc...

エラー通知の手段の一例

- int型の返り値
  - C言語のAPI
- nullableな返り値
  - fopen, java.util.Map.get(), HaskellのMaybe
- 返り値とエラー情報のタプル
  - Goのライブラリなど
- 返り値とエラーの直和型
  - HaskellのEitherなど
- オブジェクトをエラー状態にする
  - STLのstreamクラスなど
- 例外を投げる
  - Javaのライブラリ、その他近代的なものほとんど

どれを使うべきか?

実際のところ一長一短

- 返り値にエンコードするタイプは容易にチェック忘れ
  - 無視するほうが面倒なデザインのほうが望ましい
- 例外を用いるタイプは処理系のサポートが必要
  - 例外安全性との関係
  - 例外安全なプログラムを書くのはとても大変

言語・標準ライブラリのデザインの問題

どのエラー通知を使うかは、言語デザインにも関わる

- Javaなど、例外とGCをあわせて提供する
- Goなど、例外をサポートしないという選択肢

例外への批判

- 例外はコストが大きい
- 例外はコントロールフローがわからなくなる
  - goto みたいなもん、しかし…
- フローを追いかけるのはいずれにせよ大きなプログラムでは困難
  - Composabilityから性質を保証すべき（あとで）

エラー処理が言語の設計にまで影響を及ぼす理由

複数のエラーを組み合わせて使うことが難しい

- なぜ複数のエラー通知手段を用いるのが難しいのか?
  - エラー通知手段ごとに、異なる方法でエラーを捕まえなければならない から

```
{
  try {
    Hoge h = new Hoge(foo);
    Moge i = h.find(x);
    if (i == null) {
        ...
    }
  }
  catch(HogeException e) {
    ...
  }
}
```

使い分けたい時もある

- Java は全般的に例外でエラーを通知する
- Map.get()など、null で返すインターフェースもある

そもそも、何をエラーとして扱うかという話でもある

- 実際、使い分けたほうが綺麗になることもある
  - インターフェース上の制約で自由な実装ができないのは残念

エラー通知とエラーハンドリングの切り離し

そのために必要なもの

- エラーの抽象化
  - 多相的なエラー通知
    - コンテクストによって違うエラーを生成
    - nullableを返して欲しいところではnullを、例外を期待するところ では例外を投げる, etc...
  - 多相的なエラーハンドラ
    - すべてのエラー通知手段に対して動作するエラーハンドラ
    - これもコンテクストによって変化

そんなのができるのか・使えるのか?

メモリ管理との対比

- GCなんか使えない（昔）
  - ↔ GCを持つ処理系が多数（今）
- 神経すり減らしてエラー処理を書くべし（今）
  - ↔ 手でエラー処理を書くとか考えられない（私の思い描く未来）

Haskellでのアプローチ

エラーの抽象化に望まれること

- エラー処理の抜けを検出できること
- エラーに関する情報が型に現れること
- Composableであること

Composable

プログラムを組み合わせ可能であるということ

- 正しいプログラムを組み合わせても正しい

正しいとは（ここでは）

- 特定のプロパティ
  - メモリを漏らさない
  - 落ちない
  - エラーを正しく伝達する
  - などなど

これを組み合わせたプログラムにおいても保持する

例外

cf. 例外は抽象化の一例ではあるが、Composableではない

```
class Hoge {
  public void foo() {
    try {
      DB db = new DB(...);
      try {
        if (db.getRow(...)) {
          ...
        }
      }
      catch(...) {
        ...
      }
    }
    catch(...) {
      db.release();
    }
  }
```

```
}
```

Haskellでは

モナドを用いるアプローチ

- モナドは、Composable
- モナドは、ポリモーフィック
- モナドは、多相的で強く型付けされる

Error モナド　(in mtl)

```
class (Monad m) => MonadError e m | m -> e where
    -- | Is used within a monadic computation to begin exception processing.
    throwError :: e -> m a

    {- |
    A handler function to handle previous errors and return to normal execution.
    A common idiom is:

    > do { action1; action2; action3 } `catchError` handler

    where the @action@ functions can call 'throwError'.
    Note that @handler@ and the do-block must have the same return type.
    -}
    catchError :: m a -> (e -> m a) -> m a
```

エラーの送信(throwError)、エラーの受信(catchError)

Error モナド

IO例外を扱えるようにする

```
Instance MonadError IOException IO where
    throwError = ioError
    catchError = catch
```

Eitherを扱えるようにする

```
Instance Error e => MonadError e (Either e) where
    throwError             = Left
    Left  l `catchError` h = h l
    Right r `catchError` _ = Right r

Instance (Monad m, Error e) => MonadError e (ErrorT e m) where
    throwError = ErrorT. throwError
    catchError = ErrorT. catchError
```

エラーハンドラの抽象化へ

よくあるパターンを抽象化できるようになる

- エラー無視
- n回試行
- a が失敗したら b を実行

これらを組み合わせて、

- aかbのダウンロードが成功するまで10回繰り返して、失敗したエラーは無 視して（もしくはログに出して）終了するHTTPクライアント

抽象化:エラー無視

```
ign :: MonadError e m => m () -> m ()
ign m = m `catchError` (¥e -> return ())
```

受け取ったエラーを無視するだけのコード

抽象化:n回試行

```
tryN :: MonadError e m => Int -> m a -> m a
tryN n m = go n where
  go 1 = m
  go i = m `catchError` (¥e -> go (i-1))
```

失敗したらカウンタを減らして再度実行

抽象化:aが失敗したらbを実行

```
or :: MonadError e m => m a -> m a -> m a
or a b = do
  a `catchError` (¥_ -> b)
```

エラーハンドラでbを実行する

組み立てる

```
main = ign $ tryN 10 $ do
  download "http://xxx/aznn.png" `or`
  download "http://xxx/prpr.png"
```

あんなコードやこんなコードも自由自在!

技術的な課題

Composableであっても(正しくても)、記述力が十分とは限らない。

- 例えば、モナド変換子
  - catchError の実装のために、モナドの"持ち下げ"が必要
- 例
  - try :: IO a -> IO (Either e a) に、hoge :: StateT s IO a を渡したい、など

monad-control

これに対する解答が最近ようやく確立

```
class MonadTrans t => MonadTransControl t where
  data StT t :: * -> *Source

  liftWith :: Monad m => (Run t -> m a) -> t m aSource

  -- liftWith is similar to lift in that it lifts a computation from the argument monad to the constructed
  -- Instances should satisfy similar laws as the MonadTrans laws:

  -- restoreT :: Monad m => m (StT t a) -> t m a

type Run t = forall n b. Monad n => t n b -> n (StT t b)
```

ともかく、それなりに課題もある

別のアプローチ

- Verification
  - プログラムの性質を何らかの方法で検証する
- Effect Analysis
  - プログラムのEffect(IO, 例外, totality, etc...)を推論、型付けす る
  - プログラミング言語 Koka
  - ICFP2012 で今年からワークショップが開催

まとめ

- エラー処理に美しさを!
- エラーハンドラの抽象化
  - 退屈な繰り返しを避けられる
  - 適切なエラー通知手段を選べる
  - エラー処理が簡潔かつ確実になる
  - プログラムが美しくなる!

Thank you for listening!