

スタート例外!

上里 友弥 (ranha)

筑波大学

September 4, 2011

エラーハンドリング勉強会

目次

- 1 Stack を実装しよう
- 2 Haskell で Stack 実装
- 3 C で Stack 実装
- 4 C++で実装

はじめは Stack から

言わずと知れた Exceptional C++は、「例外安全」「例外中立」という2つの概念を Stack の実装を例に説明しています。
我々も今一度 Stack の実装を試みましょう。

例外安全 例外が発生しても適切に処理する

例外中立 呼び出し側にすべての例外を伝える

Stack とは?

Definition (Int 型を要素とする Stack)

- Empty
- Push(Int 型の値, スタック)
- 以上2つのみから作ることが出来るものをスタックと呼ぶ

Example

- Empty
- Push(1 , Push(2 , Empty))
- S = Push(1 , S)

ミッション

push 操作, pop 操作を実装
せよ!

Haskell で実装

Definition

```
module Stack1 where
data Stack a = Empty | Push a (Stack a)
                deriving Show
push :: (a , Stack a) -> Stack a
push = uncurry Push

pop :: Stack a -> Maybe (a , Stack a)
pop (Push v s) = Just (v , s)
pop Empty = Nothing

pop' :: Stack a -> (a , Stack a)
pop' (Push v s) = (v , s)
pop' Empty = error "うっ"
```

(push,pop) 上でのコード

Example

```
import Stack1

push2 :: Stack a -> Maybe (Stack a)
push2 stack = pop stack >>=
    \(v,s) -> return $ push v (push v s)

add :: Stack Int -> Maybe Int
add stack = pop stack >>=
    \(v,s) -> pop s >>=
    \(v',s') -> return $ v + v'
```

なんというか、全体的に厚ぼったい。

(push,pop') 上でのコード

Example

```
import Stack1
```

```
push2 :: Stack a -> Stack a
```

```
push2 stack = let (v,s) = pop' stack in  
               push v (push v s)
```

```
add :: Stack Int -> Int
```

```
add stack = let (v,s) = pop' stack in  
             let (v',s') = pop' s in  
             v + v'
```

フツーな感じ

どっちの実装が好み?

pop

- 兎に角例外を投げなけりゃ良いんだろ !?
- じゃあ失敗したことを値で表現してあげれば良いじゃん

pop'

- 空スタックへの pop が起こるのは異常事態 = 例外
- 起こっては成らない事に備える必要はねえ!!
- けどまあパタンマッチで失敗するぐらいなら例外投げる

どちらの実装が好ましい?

もっと良い実装はない?

Pop の性質満たせてる??

Theorem (Push and Pop preserve Stack)

$$\forall stack \forall v . Pop(Push(v, stack)) = (v, stack)$$

Push した後に Pop すると, Push した値と元のスタックが得られる!! という性質.

Example (pop だと型が合わない)

```
pop ( push (1 :: Int) Empty ) :: Maybe (Int , Stack Int)
```

Example (pop' だと型が合う)

```
pop' ( push (1 :: Int) Empty ) :: (Int , Stack Int)
```

やっぱり, pop' の方が好ましい?? それとも他に問題がある??

Push した値は絶対に Empty にはならない

Example (Push 色々)

- $\text{Push}(1, \text{Empty}) \neq \text{Empty}$
- $\text{Push}(x, \text{Empty}) \neq \text{Empty}$
- $\text{Push}(x, y) \neq \text{Empty}$

これは定理としてまとめることができる。

Theorem (Push never generate Empty)

$\forall \text{stack} \forall v . \text{Push}(v, \text{stack}) \neq \text{Empty}$

push の型

ところで push の型はどうなってるでしょうか??

push の型

```
Stack1> :t push  
push :: (a, Stack a) -> Stack a
```

従って, Push の逆操作の型は単純に考えると…

Definition (CoPush)

```
copush :: Stack a -> (a , Stack a)
```

copush とは Pop そのもののこと. これで, 問題点が明らかになった.

Q & A

問

Pop(Push の逆!) 操作の実装が上手くいかない理由はどこにあるでしょう?

(一応の) 答

Pop の受け取る型が,"Stack a"では Empty を考慮しなくてはならない. Pop は Push の逆なので,Push コンストラクタが"Stack a"を返すとなっているのがそもそも良くない. 実際 Push からは Empty は出てこない.

これを受けて,Stack の定義の改良とそれに伴う操作の実装を考えてみる.

改 Stack1

Definition

```
module Stack2 where

data Stack a = Empty | Pushed (NonEmptyStack a)
              deriving Show

data NonEmptyStack a = Push a (Stack a)
                      deriving Show

push :: (a , Stack a) -> NonEmptyStack a
push = uncurry Push

pop :: NonEmptyStack a -> (a , Stack a)
pop (Push v s) = (v , s)
```

改 Stack1 クライアントコード

Definition

```
{-# LANGUAGE ViewPatterns #-}  
import Stack2  
  
push2 :: Stack a -> Stack a  
push2 (Pushed (pop -> (v,s))) =  
    Pushed $ push (v , Pushed $ push (v , s))  
push2 Empty = error "うっ"  
  
add :: Stack Int -> Int  
add (Pushed (pop -> (v,Pushed (pop -> (v',_)))))  
    = v + v'  
add (Pushed (pop -> (v,Empty))) = error "うっ"  
add Empty = error "うっ"
```

push2 の型はどうすべき?

push2 の型

- `push2 :: Stack a → Stack a`
- `push2 :: NonEmptyStack a → Stack a`
- `push2 :: Stack a → NonEmpty Stack a`
- `push2 :: NonEmptyStack a → NonEmptyStack a`

この型が最も的確に表現している.

`push2 :: NonEmptyStack a → NonEmptyStack a`

Definition

```
{-# LANGUAGE ViewPatterns #-}  
import Stack2  
  
push2 :: NonEmptyStack a -> NonEmptyStack a  
push2 (pop -> (v,s)) =  
    push (v , Pushed $ push (v , s))
```

add の型はどうするべき?

add の型

- `add :: Stack Int → Int`
- `add :: NonEmptyStack Int → Int`

Example (失敗 1)

```
*Main> add (Empty :: Stack Int)
*** Exception: うっ
```

Example (失敗 2)

```
{-# LANGUAGE ViewPatterns #-}

add' :: NonEmptyStack Int -> Int
add' (pop -> (v,Pushed (pop -> (v',_)))) = v + v'
add' (pop -> (v,Empty)) = error "うっ"
```

エッ… 困った!!

何が原因?

add が欲しい Stack は, 長さが 2 以上のスタックに限られる. でも NonEmptyStack は長さ 1 以上のスタックをあらわしている.

愚直拡張

```
module Stack2_1 where

data EmptyStack a = Empty
data NotEmptyStack a = Pushed1 (Stack1 a)
                    | Pushed2 (Stack2 a)
  -- サイズ1のスタック
data Stack1 a = Push1 a (EmptyStack a)
  -- サイズ2以上のスタック
data Stack2 a = Push2 a (Stack1 a)
              | PushMore a (Stack2 a)
data Stack a = E (EmptyStack a) | NE (NotEmptyStack a)

push :: (a , Stack a) -> NotEmptyStack a
push (v , E _) = Pushed1 (Push1 v Empty)
push (v , (NE (Pushed1 s1))) = Pushed2 (Push2 v s1)
push (v , (NE (Pushed2 s2))) = Pushed2 (PushMore v s2)
```

もうこんなことは
やめましょう

長さを型で表現すれば良いですね!!

改 Stack2

Definition

```
{-# LANGUAGE GADTs , EmptyDataDecls #-}  
module Stack3 where  
  
data Z ; data S a  
  
data Stack a n where  
    Empty :: Stack a Z  
    Push  :: a -> Stack a n -> Stack a (S n)  
push :: (a , Stack a n) -> Stack a (S n)  
push = uncurry Push  
  
pop :: Stack a (S n) -> (a , Stack a n)  
pop (Push v s) = (v , s)
```

何をしているのか

一口で言えば、長さ情報が型として付与された Stack を定義した。

Definition (Empty :: Stack a Z)

Empty は要素の型を a とする、長さ Z (すなわち 0) のスタックである。

Definition (Push :: a → Stack a n → Stack a (S n))

Push は要素の型 a の値と、長さ n のスタックを取り、長さ $S(n)$ (すなわち $n+1$) のスタックを作る

単にそれだけのこと。しかし次のページの例が示すように、型チェッカが口うるさく成る事を利用したコードが書ける。

改 Stack2 クライアントコード

Definition

```
{-# LANGUAGE GADTs #-}  
  
import Stack3  
  
push2 :: Stack a (S n) -> Stack a (S (S n))  
push2 stack = let (v , s) = pop stack in  
               push (v , push (v , s))  
  
add :: Stack Int (S (S n)) -> Int  
add stack = let (v , s) = pop stack in  
             let (v' , s') = pop s in  
             v + v'
```

良い事尽くめではない

Example (Homogeneous List)

```
[ Empty :: Stack Int Z ,  
  push (1 , Empty) :: Stack Int (S Z) ] :: ???
```

どうする?

Definition (erase 関数)

```
erase :: Stack a n -> SigT (Stack a)  
erase stack = ExistT stack
```

Example (Homogeneous List 再訪)

```
[ erase (Empty :: Stack Int Z) ,  
  erase (push (1 , Empty) :: Stack Int (S Z)) ] :: [ SigT (Stack a) ]
```

erase 関数実装

Definition (erase 関数実装)

```
{-# LANGUAGE GADTs , RankNTypes , KindSignatures #-}  
import Stack3  
  
data SigT (p :: * -> *) where  
    ExistT :: p x -> SigT p  
-- SigT p <-> exists x , p x  
  
-- erase :: Stack a n -> (exist n . Stack a n)  
erase :: Stack a n -> SigT (Stack a)  
erase stack = ExistT stack
```

抵抗があれば、サイズ無しの Stack を定義して単純にそれへの変換関数を書けば良い。

Haskell 編まとめ

- ガッチリした定義を与えれば適当なインタフェースになる
 - 例外を投げずに済みました
 - ゆるゆるにして例外を投げるのも手
 - コピーと生成とその例外に気を揉む必要はない
-
- 25 Haskell 流の例外処理を学ぶ
 - 28 例外やエラーに対する処理能力を加えるエラー・モナド

C で Stack 実装

案 1

```
struct Stack;  
struct Push {  
    int i;  
    struct Stack *p;  
};
```

案 2

```
struct Stack {  
    int used;  
    int *array;  
};
```

取りあえず Empty の表現が楽な後者を採用。

制約

縛りプレイ

- Stack は整合性の取れた状態を保ち続ける.
- malloc で確保したメモリは, 必ず free されなければならない.
- Stack 内部で何か良く無い事が起こったら隠蔽せず, クライアント (ユーザ) に伝える

取りあえず実装

↓ のコード, 縛りに反する (可能性はある)??
view stack_impl1.c

memcpy , free , malloc

```
void* memcpy(dst,src,size)
```

src から size バイト dst を起点にコピー. まあ多分問題無い. 返値は dst

```
void free(ptr)
```

malloc が返した値を free する分には問題無い. free(NULL) も問題無い.

```
void* malloc(size)
```

エラーの場合 **NULL が返る**. errno は ENOMEM に. malloc(0) でも (そしてその結果を free しても) 問題無い.

stack_impl1.c は縛りのもとで問題を抱えまくっている.

- malloc の失敗を呼び出し側に通知しない隠蔽体質
- top で困ったら abort する隠蔽体質 (ただし abort しても free 縛りには反しない)
 - Maybe を書くのが面倒臭い **言語が悪い**
- デルアルで正常化するまでに, 不整合なスタックが「事実上」存在しているし使っちゃうかも
- デルアルで正常化しない謎スタックを destruct すると面白い
 - コンストラクタが無い **言語が悪い**
- pushloop でスタックが溢れて異常死, 結果 free 出来ない
 - スタックが溢れるのが悪い **言語が悪い**

言語が悪い
ではなく
GCC を活用する

Maybe どうする?

```
struct Maybe {  
    T t; /* 要素型 */  
    uint8_t is_Some; /* Some か Nothing か */  
};
```

書く気がしないし, 使う気が起きない.

なら書かなければ良い. かつ, 隠蔽体質は改善したい.

⇒ 返値で場合分けして計算「ではなく」場合分けされた計算そのものを用いる.

成功時と失敗時の計算を作る

```
void push(int v, struct Stack * const s,
          void (*S)(struct Stack),
          void (*F)(void))
{
    int *tmp = malloc(sizeof(int) * (s->used + 1));
    if(tmp == NULL) F();

    forward(tmp, s->array, sizeof(int) * s->used);

    tmp[s->used] = v; ++s->used; s->array = tmp;

    S(*s);
}
```

```
void pop(struct Stack s,  
         void (*S)(int), /* 成功時 pop操作でスタックから  
消えた値を受ける */  
         void (*F1)(void), /* スタックが空の時 */  
         void (*F2)(void)) /* malloc 出来ない時 */ {  
    if(s.used == 0) F1();  
    else {  
        void cont(int v) {  
            ...  
            S(v);  
        }  
        top(s, cont, NULL);  
    }  
}
```

gcc の extension nested-function を用います. スコープ的にも楽!!

view stack_impl2.c

スタックオーバーフロー対策

いわゆる separate(alternate) signal stack を用いる. (以下参考)

- <http://www.gnu.org/s/hello/manual/libc/Signal-Stack.html>
- スタックオーバーフローのハンドリング
- MSDN _resetstkoflw

SEGV 対策取りあえず版

view stack_impl3.c

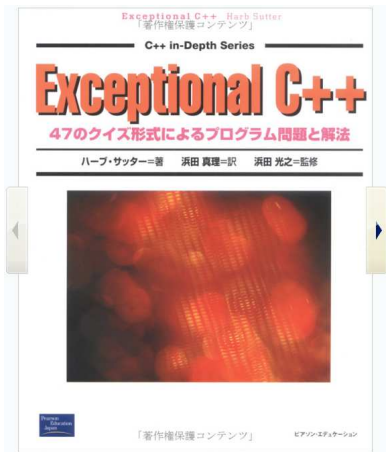
もし例外が使えたら…

- 成功時と失敗時の計算を一々作らなくて良い
- 失敗時の情報伝達が割とまとも
- 規格に入っている程度の標準手法が使える嬉しさ

C++でどう実装するのか!?

Stack を実装しよう
Haskell で Stack 実装
C で Stack 実装
C++で実装

続きは Exceptional C++で!



(Amazon.co.jp さんから画像拝借)

発表のまとめ

なし