

# 合成モナドのための call/ccに対する改善の提案

筑波大学

よねざわ たくお

米澤 拓央

この辺は何書いても後ろの人から見えない☹️。

# Haskellのcall/cc

## 重大な

## Haskeller騒然

# 設計ミスか!?

### オペレータ

## get/ccの意外な正体とは!?

プログラミング言語Haskellにおいて継続を扱うオペレータであるcall/cc (HaskellではcallCCと表記される)にモナド変換子を使った場合に

問題が発生すること  
が判明した。

Haskellではエフェクト(例外機構や状態や継続の操

作を表すためにモナドを使っている。そして複数のエフェクトを同時に使う際にはモナド変換子を使ってモナドを合成している。この

## モナド変換子に 対応できず

が現れる。しかしの操作は実はbind演算子であるvmapややっていることは同じである。むしろモナド変換子に対応していない、め劣化  
ピーと  
てもよ  
call/cc

際、ベースとなるモナドの値を合成したモナドの値に変換するためにliftという演算子を用いているが、`lift`が変換できるのはあくまで値だけである。そのため、モナドを扱う関数については関数合成によって返り値の型を変えることはできないが、引数の型まで変換することができない。

この典型例がcallCCだ。

現在の実装では、この問題に対してはcallCCをオーバーロードし、既知の組み合わせ全てに対しての実装を個別に手作業で与えること

によって対応している。しかし、これではモジュール性が崩れてしまう。

しかし、そもそもこの問題を考えてみるとこの問題は本質的な問題ではないことが分かる。

- ① 継続を捉えて関数にする。
- ② その継続を引数として関数を呼び出す。

では、この本質的な部分のみを抜き取らないだろうか。それがget/ccである。get/ccは自身の継続を関数として返す関数である。しかし、その際、関数は継続を返すと動作とその継続がばれたときの値を返すという二つの動作をしなければならぬ。そこで、直積、`Haskell@Either`を利用する。get/ccはまず最初に自分の継続Rightで包んだ値を返す。そしてその継続が呼び出されたとき、get/ccは今度は継続に渡された値を包んで返す。これによりget/ccは型を取らなくなる。型の問題は発生なくなる。

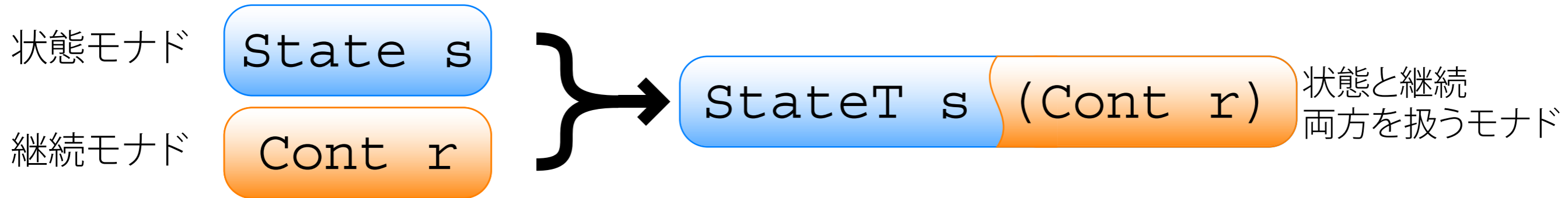
では、`lift` get/ccはどのよう実装できるのだろうか、`lift` get/ccはcallCCを使って実装できる。しかもわずかに二定義で済んでしまう。

さらに、`lift` get/ccを使うとcall/ccを一般化した定義ができる。これは、継続モナドを演算子が与えら

# モナド変換子

モナド: エフェクト(例外・状態・継続の操作)を表す。

モナド変換子: 複数のエフェクトを同時に使うためにモナドを合成する機構。



`lift : Cont r a -> StateT s (Cont r) a`

# 問題点

`callCC` :: ( (a -> `Cont r` b) -> `Cont r` a) -> `Cont r` a

`lift . callCC` :: ( (a -> `Cont r` b) -> `Cont r` a) -> `StateT s (Cont r) a`

同様の問題が例外ハンドラなど、  
モナドの型を引数の型に含む関数で起きる。

# 分析

call/ccの仕事:

```
callcc (\c -> ...)  
>>= (\x -> ...)
```

1. 継続を関数として捉える。
2. その継続を、関数に渡す。  
(ここで型の問題が起きる。)

# 分析

call/ccの仕事:

```
callcc (\c -> ...)  
>>= (\x -> ...)
```

1. 継続を関数として捉える。
2. その継続を、関数に渡す。  
(ここで型の問題が起きる。)

しかし、2はbind演算子>>=とやってることは同じ。  
call/ccの仕事は1のみが本質である。

# 提案: get/cc

```
callCC (\c -> ...)
```

```
>>= (\v -> ...)
```

継続

継続が渡される部分

継続が渡される部分

```
getCC >>= (\x ->
```

```
case x of
```

```
(Right c) -> ...
```

```
(Left v) -> ...
```

継続

vの型

cの型

```
getCC :: Cont r (Either a (a -> Cont r b))
```

```
getCC = callCC (\c0 ->  
    return (Right (c0 . Left))
```

# 提案: 一般化call/cc

```
genericCallCC :: (Monad m)
              => (forall d. Cont r d -> m d)
              -> ((a -> m b) -> m a) -> m a
```

```
genericCallCC cast body
  = (cast getCC) >>= (\x ->
    case x of
      (Right c) -> body (cast . c)
      (Left v)  -> return v
```

継続モナドの値を別のモナドの値に変換する関数を与えると  
そのモナド用のcallCCを返す。

# まとめ

Haskellのcall/ccでは複数の種類のモナドを組み合わせて使うときに制約が出る。

解決するためにcall/ccの本質的な部分だけを抜き出した演算子**get/cc**を提案した。

```
getCC :: Cont r (Either a (a -> Cont r b))
getCC = callCC (\c0 ->
                return (Right (c0 . Left)))
```

以上。  
米澤 拓央  
@筑波大学