Vol.2018-SE-198 No.3
2018/3/9

Haskellプログラムへの状態モナドの自動挿入

堀内 哲熙[1,a)]　大久保 弘崇[1,b)]　粕谷 英人[1,c)]　山本 晋一郎[1,d)]

概要：プログラミング言語 Haskell は，全ての関数が参照透過性を保つため，副作用が生じない．一方で変数への再代入ができないため，手続き型言語のように状態を扱うことができない．Haskell で状態を扱う際は，標準化された方法として，State モナド [1] が提供されている．しかし，State モナドの使用を想定せずに設計されたプログラムの開発途中に State モナドを使用する必要が生じた場合，それに起因して作成済みのコードに，多くの修正が必要となりうる．本論文では，State モナド挿入時の修正箇所の特定及び修正を自動化する手法を提案する．また，提案手法を実装し，実際のプログラムを変換する実験を行なった．

1. はじめに

1.1 背景

プログラミング言語 Haskell は，関数型言語でありその中でも純粋関数型言語に属する．関数型言語はプログラムの全てが関数で構築され，更に純粋関数型言語はその関数全てが，同じ入力に対しては同じ出力を返すという参照透過性を保つため，副作用が生じないという特徴がある．この特徴により Haskell は高い保守性を持ち，不具合を局所化できる等のメリットを持つ．一方で，その特徴がデメリットになる事もある．

Haskell は参照透過性を保つため，変数への再代入を禁止している．この言語仕様により Haskell では状態を使用する際，手続き型言語の様には状態の更新が行えない．Haskell で状態を扱う場合，専用の機能を備えた State モナド [1] を用いるのが一般的である．State モナドを用いると Haskell でも状態を容易に扱うことができる．

しかし，State モナドを用いてプログラミングをする場合に問題となる事がある．それは，State モナドを使用することが想定されていないような既存のプログラムに導入する際，状態を使用したい箇所がプログラムの一部だけでも，その他の多くの部分を書き換えなければならないという事である．プログラムの規模が大きくなるにつれて，修正が必要な箇所も増えて行き，プログラムを手作業で書き換えるのは困難な作業になる．

[1] 愛知県立大学情報科学部
School of Information Science and Technology, Aichi Prefectural University
[a)] t.horiuchi@yamamoto.ist.aichi-pu.ac.jp
[b)] ohkubo@ist.aichi-pu.ac.jp
[c)] kasuya@ist.aichi-pu.ac.jp
[d)] yamamoto@ist.aichi-pu.ac.jp

```
1 newtype State s a =
2       State {runState :: (s -> (a, s))}
```

図 1　State モナドの型定義

1.2 目的

本論文では，既存の Haskell プログラムに State モナドを挿入する際，修正が必要となる箇所の特定及び修正を自動化する手法を提案する．また，提案手法を実装し，プログラマが Haskell で状態を扱うときの負担を軽減する事を目的とする．

1.3 本論文の構成

本論文では，第 2 章で State モナドについて述べ，第 3 章で State モナド挿入時のプログラムの修正箇所の特定・修正手法について述べる．第 4 章で提案手法の実現方法とその評価を行い，第 5 章ではまとめと今後の課題について述べる．

2. State モナド

本章では，State モナドの型定義，初期状態の与え方，状態の扱い方等について述べる．

2.1 State モナドの型定義

State モナドは `Control.Monad.State`モジュールにより提供される．型定義を**図 1** に示す．State 型コンストラクタは，2 つの型引数 s と a を取る．s は状態の型，a は State モナドを扱う関数の値の型である．

`Control.Monad.State`モジュールでは，状態を操作するための 2 つの関数 get と put を備えた MonadState 型クラスを提供している．get 関数は現在の状態の取得に用いる

```
1  import Control.Monad.State
2
3  main = print $ evalState count3 5
4
5  count3 :: State Int [Int]
6  count3 = do
7    c1 <- incrState
8    c2 <- incrState
9    c3 <- incrState
10   return [c1, c2, c3]
11
12 incrState :: State Int Int
13 incrState = do
14   s <- get
15   put (s+1)
16   return s
```

図 2 incrState

関数であり，put 関数は状態の更新に用いる関数である．

State モナドを用いた関数の計算を行うには，runState関数を用いる．runState関数は、State アクションと状態の初期値を引数にとり，アクションを実行してその結果と最終状態の対を返す．

```
runState :: State s a -> s -> (a, s)
```

また，利便性のために，結果または最終状態を取り出す

```
evalState act ini = fst (runState act ini)
execState act ini = snd (runState act ini)
```

という関数も提供されている．

2.2 State モナドの使用例

実際に State モナドを使用した例を**図 2** に示す．

図 2 は main関数，count3関数，incrState関数から構成される．incrState関数は，状態に 1 を加えた値を新しい状態とし，以前の状態を返り値として返す．count3関数は，incrState関数を 3 度呼び出し，その返り値をリストにして返す．

main関数で，初期状態を 5 として count3関数を実行すると出力は以下になる．

```
> main
[5,6,7]
```

初期状態 5 に対して，incrState関数が呼ばれた回数だけ，値に 1 が加えられていることが確認できる．

2.3 既存プログラムへの State モナドの挿入

この節ではサンプルプログラムを用いて，State モナドの挿入がプログラムに与える影響について解説する．

2.3.1 サンプルプログラム

図 3 に示すプログラムは，main関数，showCount関数，counter関数の 3 つの関数から構成されている．main関数は showCount 関数の返り値を標準出力へ出力する．showCount 関数は，counter関数から Int型のカウントを 2 回受け取り，“c1 is ◯◯, c2 is △△”というように，カウントを組み合わせた文字列を返す．counter関数は，最初に呼び出された時は 1，2 回目の呼び出しでは 2，というように呼び出された回数を返す．

しかし，counter関数の実装は未完成であり，常に 0 を返す関数となっている．意図通りに counterの機能を実現するためには「現在のカウントの値」という状態を扱えるようにする必要がある．そこで counter関数に State モナドを挿入し，状態を扱えるように修正するとプログラムは**図 4** のようになる．この時の実行結果は以下のようになる．

```
"c1 is 1, c2 is 2"
```

注目すべき点は，状態を扱う関数は counter関数だけであるにも関わらず，showCount関数と main関数のコードにも変更が及んでいる所である．

showCount関数は，counter関数がアクションへ変化したことにより，counter関数の返り値を “<-を用いて変数へ束縛する必要が出てきた．これにより showCount関数自身もアクションへと変化する．

また，main関数では，showCount関数が State モナドのアクションへ変化したため，単純な呼び出しのかわりに，evalStateを用いて状態計算の起動を行う必要が出てきた．

このように，既存のプログラムに State モナドを挿入すると，状態を扱いたい関数だけでなく他の関数の実装も修正が必要になりうる．

3. 提案手法

本章では，既存のプログラムに State モナドを挿入する際，修正が必要となる箇所の特定及び修正を自動化する手法について述べる．

3.1 プログラムへの状態挿入手続き

この節では，既存のプログラムへ状態を挿入する手続きについて述べる．この手続きは**図 5** で表される Haskell のサブセットを対象とする．ここで，*pat* はパターン，*lit* はリテラルである．{ } で囲まれた要素は 1 回以上の繰り返しを表す．

3.1.1 入出力と制約事項

手続きは，Haskell プログラムの他に，状態の型 S，状態を挿入したい関数群 f_{state}，状態計算を起動する関数群 f_{run}，各 f_{run} の状態計算の初期値 *init* を入力としてとり，f_{state} の関数が状態を持つように修正されたプログラムを

```
1  main :: IO ()
2  main = print showCount
3
4  showCount :: String
5  showCount =   "c1 is " ++ (show c1) ++
6              ", c2 is " ++ (show c2)
7    where c1 = counter
8          c2 = counter
9
10 counter :: Int
11 counter = 0
```

図 3　状態挿入前のプログラム

```
1  import Control.Monad.State
2
3  main :: IO ()
4  main = print $ evalState showCount 0
5
6  showCount :: State Int String
7  showCount = do
8     c1 <- counter
9     c2 <- counter
10    return $   "c1 is " ++ (show c1) ++
11             ", c2 is " ++ (show c2)
12
13 counter :: State Int Int
14 counter = do c <- get
15              put (c+1)
16              return (c+1)
```

図 4　状態挿入後のプログラム

出力する．

また，この手続きを適用できるのは以下の制約事項を満たすプログラムである．

(1) 関数適用の左辺は関数名であるか関数適用である．すなわち，カリー化以外で関数を得る計算を禁止する．
(2) let 式は再帰，相互参照のような循環を持たない．
(3) let 式で局所関数を定義していない．
(4) ラムダ式の本体は状態を挿入する関数の呼び出しを起こさない．

3.1.2　変換手順

修正範囲の特定

入力されたプログラム内の，関数の呼び出し関係を表すコールグラフを用いて修正範囲の特定を行う．関数名を節，呼び出し関係を辺とする有向グラフを作成し，これをコールグラフとする．

この時，f_{run} から f_{state} までの経路上にある関数 (ただし，f_{state} を含み，f_{run} を含まない) が修正対象となる関数集合 f_{fix} である．

ここで、f_{fix} に含まれる関数は f_{fix} にも f_{run} にも含まれない関数から呼び出されていてはならない．そのような呼び出し経路がないように f_{run} を設定する必要がある。

宣言　*decl*　::=　*pat* = *exp*
　　|　*String pat* = *exp*

式　*exp*　::=　let { *decl* } in *exp*
　　|　if *exp* then *exp* else *exp*
　　|　case *exp* of { *pat* -> *exp* }
　　|　do { *stmt* }
　　|　*exp exp*
　　|　\{*pat*} -> *exp*
　　|　*lit*

文　*stmt*　::=　*pat* <- *exp*
　　|　*decl*
　　|　*exp*

図 5　Haskell サブセットの構文

f_{fix} への State モナドの導入

f_{fix} に含まれる関数を，純粋関数から State モナドアクションに変更する．

これは，関数の最終的な戻り値が tであるとき，それを State S tに変更することを意味する．そのように型シグチャを変更する．また，State モナドアクションとなるように関数本体を変更する．これは次に述べる手続き *addState* により行う．

たとえば，f_{fix} に次のような関数 g が含まれ，

```
1  g :: a -> b -> t
2  g x y = e
```

式 e を *addState* で変更した結果が式 e' であるとき，g の定義を

```
1  g :: a -> b -> State S t
2  g x y = e'
```

と変更する．

状態計算の起動を行う関数の修正

f_{run} の中で，f_{fix} に含まれる関数の呼び出し部分を，State モナドの起動に修正する．State モナドの起動には，修正前後の型を合わせるために，evalStateを用いる．

すなわち，f が f_{fix} に含まれる関数である時，f_{run} 内の関数適用 $f\ e$を evalState ($f\ e$) *init*へと修正する．この時，$f\ e$と evalState ($f\ e$) *init*の型は同一となる．

3.2 式の修正

addState は t型の式を State S t型のアクションへと変換する手続きである．

3.2.1 入出力

addState は t型の式を入力としてとり，State S t型のアクションを出力する．

3.2.2 変換手順

addState では，変換対象となる式 e の構造により場合分けを行う．以後，*addState*(e1) = e2という表記は，Haskell の式である e_1 を本手続き *addState* で変換した結果が e_2 になることを表す．

部分式を持たないリテラルや変数については，式を State モナドに包むことで変換が完了する．複合式については，その部分式が f_{fix} に含まれる関数を呼び出しているかどうかで取るべき対応が変わる．以後，f_{fix} に含まれる関数を呼び出していない式を (State モナドについて)『純粋』な式と呼ぶ．

部分式全てが純粋である複合式 e は，全体として純粋なので *addState*(e) = return eとなる．純粋でない部分式を持つような複合式については，全体を do 式に置き換え，純粋でない部分式から値を “<-” を用いてスコープ内でフレッシュな変数 x に束縛し，部分式を x に置き換える．

以下，式の種類ごとに具体的な変換方法を述べる．

リテラル

式 e がリテラルである場合は，e を State モナドへ包む必要がある．よって *addState*(e) = return eとなる．

変数

リテラル式の変換と同様に，式 e が変数の場合 *addState*(e) = return eとなる．

case 式

case 式では，条件式が純粋な式かどうかで場合分けを行う．**図 6** を変換前の式 e とする．また，*addState*(e_0) = e'_0，*addState*(e_1) = e'_1，*addState*(e_2) = e'_2，*addState*(e_n) = e'_nである．

条件式が純粋な場合，*addState*(e) は分岐先の式全てに *addState* を適用した，**図 7** になる．

条件式が非純粋な場合，更に全体を do 式へ置き換え，条件式に *addState* を適用した上で，値を変数へ取り出す必要がある．*addState*(e)は**図 8** に変換される．

if 式

if 式 `if b then` e_1 `else` e_2 は**図 9** に示す case 式と等価なので，*addState* でも同様に変換する。

let 式

let 式は，全体を do 式に置き換え，束縛を変数の依存性を考慮しながら do 式へ移動する．この時，束縛の右辺が純粋なものについては “=” で，非純粋なものについては “<-” で束縛を行う．let 式の本体は *addState* によって変換したのち，do 式の最後に追加する．

```
1 case e0 of
2   p1 -> e1
3   p2 -> e2
4   ⊢
5   pn -> en
```

図 6 変換前の case 式

```
1 case e0 of
2   p1 -> e'1
3   p2 -> e'2
4   ⊢
5   pn -> e'n
```

図 7 変換後の case 式 (条件式が純粋な場合)

```
1 do x0 <- e'0
2    case x0 of
3      p1 -> e'1
4      p2 -> e'2
5      ⊢
6      pn -> e'n
```

図 8 変換後の case 式 (条件式が純粋でない場合)

```
1 case b of
2   True  -> e1
3   False -> e2
```

図 9 if 式と等価な case 式

```
1 let x1 = e1
2     x2 = e2
3     ⊢
4     xn = en
5 in  e0
```

図 10 変換前の let 式

```
1 do x1 <- e'1
2    let x2 = e2
3    ⊢
4    let xn = en
5    e'0
```

図 11 変換後の let 式

図 10 を e とすると，*addStateE*(e)は**図 11** へ変換される．ただし，e_2，e_n は純粋な式で，e_1 は非純粋な式であり *addState*(e1) = e'_1である．更に，*addState*(e) = e'_0である．

束縛を do 式へ移動する際，x1 が e_1 内で参照されていたり，x1 が e_2 内で参照されており，かつ x2 が e_1 内で参照されているような場合，正しく移動することができない．このため，制約事項 (2) を設けている．

関数適用

関数適用は，引数全てが変数であるかどうかで対応が異なる．

引数全てが変数である場合，$f\ x_1\ x_2\ \ldots\ x_n$を変換前の式 e とすると，f が f_{fix} に含まれる関数のとき，e は既に State モナドに包まれた値となっているため $addState(e)$ = eとなる．f が f_{fix} に含まれない関数のとき，e は純粋な値となるため State モナドに包む必要がある．そのため $addState(e)$ = return eとなる．

引数全てが変数でない場合，一般に関数適用は引数の計算を局所変数に束縛しても結果は変化はしないため，$f\ e_1\ e_2\ \ldots\ e_n$を e とすると，e は let 式を用いて let { x_1 = e_1; x_2 = e_2; ... x_n = e_n } in $f\ x_1\ x_2\ \ldots\ x_n$ のように引数全てが変数である式 e' に置き換えることができる．引数全てが変数でない関数適用について $addState(e)$ = $addState(e')$ とする．このとき e' は let 式であるため，右辺の $addState$ では先に述べた変換が適用され，let 式の本体は引数全てが変数であるためこれに対する $addState$ では上述の変換が適用される．

ラムダ式

制約事項 (1) の元では、ラムダ式は純粋な値としてしか意味を持たない．よって $addState$(\x -> e) = return (\x -> e)となる．この e は制約事項 (4) により純粋である．

e が純粋でない場合，e を $addState(e)$ の結果に置き換えて状態計算をこのラムダ式の中と外で連続させることは，そのようなモナドアクションを呼び出す方法が制約事項 (1) より存在しないため，変換することはできない．

その上，(\x -> e) :: a -> b, $addState(e) = e'$ のとき，(\x -> e') :: a -> State S bであり，両者は型が異なり，手順 $addState$ は t 型の式を State S t型に変換するという範囲を逸脱する．

do 式

StateT モナド変換子を用いてモナドを重ねることもできるが，今回は対象外とする．

4. 提案手法の実現

4.1 ツール化

3 章の提案手法に基づき，プログラムに State モナドを自動で挿入するツールを実装した．

本ツールは，修正対象の Haskell プログラム，状態を挿入したい関数 (f_{state})，状態の型 (S)，初期値 ($init$) を受け取り，State モナド挿入後のプログラムを出力する．

構文解析によりプログラムの抽象構文木を生成し，手法に基づいて抽象構文木を修正し、最後に修正した構文木を Haskell のソースコードへ復元することで状態挿入後のプログラムを出力する．(状態計算の起動を行う関数は main と仮定している。)

Haskell プログラムから抽象構文木の作成や，抽象構文木から Haskell ソースコードへの復元には [2] を用いた．

本ツールに投入するプログラムは状態を挿入したい関数の実装がスタブ e になっているいわゆる未完成の状態である．本ツールを用いて変換した後，ユーザーは return eと変換されたスタブを状態を扱う本来の実装に書き直すことでプログラムを完成させる．

4.2 サンプルプログラムの変換

作成したツールを用いて，実際にサンプルプログラムの変換を行い，変換前との比較を行う．

4.2.1 ハノイの塔のロギング

図 12 は n 次のハノイの塔を解く hanoi関数を含んだプログラムである．

円盤は Int 型のリストで表し，数が小さいほど小さい円盤である．これと，A，B，Cの杭を表す Pole型とのタプルで，一つの塔を表す Tower型が作られる．また，Tower型のトリプルを Towers型とする．

hanoi関数は次数 n と Towersを受け取り，Aの杭にある円盤を，Cの杭に移動する．以下では n は 3 と固定する．実行結果を以下に示す．

```
> main
((A,[]),(B,[]),(C,[1,2,3]))
```

円盤が Cへ移動されていることは確認できるが，途中経過が一切分からず正しく動作しているのかは分からない．そこで，Towersの状態を 1 ステップごとにロギングする事を考える．

円盤の移動が実際に行われるのは，move関数を呼び出した時である．move 関数を呼び出し結果を，リストに順に追加していくことでロギングは可能となる．

そこで，状態を挿入したい関数を move，状態の型を [Towers]，初期値を [start]として本ツールを適用すると，出力は**図 13** になる．なお，可読性のため，改行とインデント，空白の削除などプログラムの実行結果に影響の無い部分のみ修正をしてある．

move関数に渡される Towersは杭の位置が A，B，Cの順に揃っていないので，杭の位置を A，B，Cの順に揃えたあとに状態に追加する必要がある．

これを行う関数 addLog 関数を**図 14** に示す．この addLog 関数を用いて，ログを状態に追加していくように，move関数を**図 15** のように修正する．

最後に，今回必要なのは状態計算の結果ではなく，状態の方であるので，main関数での状態計算の起動方法を evalStateから execStateへ変更する．また，表示結果を見やすくするために printの前に mapM_関数を置き，**図 16** の

```
module Main where

import Control.Monad.State

data Pole = A | B | C
   deriving (Show, Eq)
type Tower = (Pole, [Int])
type Towers = (Tower, Tower, Tower)

main :: IO ()
main = do
  let n = 3
  let start = ( (A, [1..n]),
                (B, []),
                (C, []) )
  print $ hanoi n start

hanoi :: Int -> Towers -> Towers
hanoi 1 t = move t
hanoi n (a,b,c) =
 let
   (a1,c1,b1) = hanoi (n-1) (a,c,b)
   (a2,b2,c2) = move (a1,b1,c1)
   (b3,a3,c3) = hanoi (n-1) (b2,a2,c2)
 in (a3,b3,c3)

move :: Towers -> Towers
move ((pa, a:as), bs, (pc,cs)) =
  ((pa, as), bs,(pc, a:cs))
```

図 12　プログラム hanoi

ようにする．

ここまでの修正を行うと実行結果は以下のようになる．

```
> main
((A,[]),(B,[]),(C,[1,2,3]))
((A,[1]),(B,[]),(C,[2,3]))
((A,[1]),(B,[2]),(C,[3]))
((A,[]),(B,[1,2]),(C,[3]))
((A,[3]),(B,[1,2]),(C,[]))
((A,[3]),(B,[2]),(C,[1]))
((A,[2,3]),(B,[]),(C,[1]))
((A,[1,2,3]),(B,[]),(C,[]))
```

結果は上に行くほど後の状態になり，下に行くほど初期状態に近づく．杭 Aにあった円盤が杭 Cへ移動されるまでの手順が正しく記録されており，プログラムが正しく動いている事が確認できた．

このように，State モナドを用いるとロギング等も行えるようになる．今回のサンプルはプログラムの規模が小さいため，手動で修正した時と比べ修正箇所が少なくなるというメリットが感じづらいが，プログラムの規模が大きくなった時，手動で修正を行うと修正箇所は膨大となるため，

```
module Main where

import Control.Monad.State

data Pole = A | B | C
   deriving (Show, Eq)
type Tower = (Pole, [Int])
type Towers = (Tower, Tower, Tower)

main :: IO ()
main = do
 let n = 3
 let start = ( (A, [1..n]),
               (B, []),
               (C, []) )
 print $ (evalState (hanoi n start) [
     start])

hanoi :: Int -> Towers -> State [Towers
    ] (Towers)
hanoi 1 t = move t
hanoi n (a, b, c) =
  do
  (a1,c1,b1) <- hanoi (n-1) (a,c,b)
  (a2,b2,c2) <- move (a1,b1,c1)
  (b3,a3,c3) <- hanoi (n-1) (b2,a2,c2)
  return (a3,b3,c3)

move::Towers -> State [Towers] (Towers)
move ((pa, a : as), bs, (pc, cs))
  = return ((pa, as), bs, (pc, a : cs))
```

図 13　プログラム hanoi(ツール適用後)

```
addLog :: Towers -> State [Towers] (
    Towers)
addLog (a, b, c) = do
 log <- get
 put ((getPole A, getPole B, getPole C)
     :log)
 return (a, b, c)
  where
    getPole p =
      if p == (fst a) then a
      else if p == (fst b) then b
      else c
```

図 14　addLog 関数

このツールを使用するメリットは大きいと言えよう．

4.2.2　mySum

図 17 は標準ライブラリの sum関数を再定義するものである．初期状態を 0 とし，mySumCore関数内で状態を取得し，引数で受け取った値と状態の値の和を新たな状態とし，この値を返り値とする事で sumの実装を行う．

現在この mySumCore関数は常に 0 を返す関数となってお

```
move :: Towers -> State [Towers] (
    Towers)
move ((pa,a : as),bs, (pc,cs)) = do
 let result = ((pa,as),bs,(pc,a : cs))
 addLog result
```

図 15　hanoi の move 関数 (修正後)

```
main :: IO ()
main = do
 let n = 3
 let start = ((A,[1..n]),(B,[]),(C,[]))
 mapM_ print $ (evalState (hanoi n
     start) [start])
```

図 16　hanoi の main 関数 (修正後)

り，実行すると以下のようになる．

```
> main
0
```

図 17 に対し，状態を挿入したい関数を mySumCore，状態の型を Int，初期値を 0 として本ツールを適用すると出力は図 18 のようになる．この時点ではプログラムの出力は変わらない．しかし，mySumCore関数内で状態が扱えるようになったため，実装を図 19 のように書き換える事で出力は以下のようになり，プログラムが完成する．

```
> main
6
```

5. おわりに

5.1 まとめ

本論文では，既存のプログラムに State モナドを挿入する際，修正が必要となる関数の特定方法，及びその関数の修正方法を提案した．

本手法では，プログラム中で “状態を使用する関数”，“初期状態を与える関数”の二つを用いて，State モナド挿入の影響を受ける関数を特定する．

関数の修正では，Haskell のサブセットとなる構文を定義し，変換手法を提案した．本手法は関数右辺の式に着目し，その構造によって異なる変換を行う．

また，提案手法に基づき，プログラムに State モナドを自動挿入するツールを作成し，サンプルプログラムを変換する実験を行なった．

5.2 今後の課題

今後の課題としては，変換対象のプログラムに与えている制約を緩める事があげられる．現在，State モナドを挿入する関数，及びその影響を受ける関数には，3.1.1 節で述

```
import Control.Monad.State

main :: IO ()
main = do print $ mySum [1,2,3]

mySum :: [Int] -> Int
mySum []       = let sum = mySumCore 0
                 in sum
mySum (n:ns)   = let sum = mySumCore n
                 in mySum ns

mySumCore :: Int -> Int
mySumCore n = n
```

図 17　プログラム mySum

```
import Control.Monad.State

main :: IO ()
main = do print $ (evalState (mySum [1,
    2, 3]) 0)

mySum :: [Int] -> State Int (Int)
mySum []
  = do sum <- mySumCore 0
       return sum
mySum (n : ns)
  = do sum <- mySumCore n
       mySum ns

mySumCore :: Int -> State Int (Int)
mySumCore n = return n
```

図 18　プログラム mySum(ツール適用後)

```
mySumCore :: Int -> State Int (Int)
mySumCore n
  = do sum <- get
       put (sum + n)
       return (sum + n)
```

図 19　mySumCore 関数 (修正後)

べている制約がある．現在の制約では，関数を変数へ束縛する事が許されておらず，これは Haskell を扱う上では致命的な制約であると思われる．この制約を解決する事でさらに多くの Haskell プログラムを変換できるようになる．

謝辞　本研究は JSPS 科研費 JP15K00488 の助成を受けたものである．

参考文献

[1] Control.monad.state. https://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-State.html.
[2] haskell-src-exts: Manipulating haskell source: abstract syntax, lexer, parser, and pretty-printer. https://hackage.haskell.org/package/

`haskell-src-exts-1.20.1`.