Vol.2019-SE-201 No.5
2019/3/7

Haskellを対象とした先行評価を模したトレースの実現手法

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

概要：Haskell における実行トレースでは, 遅延評価に即した結果が提示される. 遅延評価に即したトレースは, 関数本体のトレースの中に引数のトレースの断片が入り組むよう表示され, ある式が最終的にどのような式になったかという評価関係がわかりづらい. このことは, どの式の評価中にエラーが発生したかの特定の妨げとなる. 一方, 先行評価の言語では引数のトレースが最終的な評価結果まで表示されたあと関数本体のトレースが表示されるので評価関係がわかりやすい. そこで本論文では,Haskell における, より評価関係のわかりやすいトレースの提示を目標とし, 遅延評価に即したトレースを先行評価でのトレースに擬似的に表示する手法を提案する. そして Haskell での実行トレースを行うツールである Hat をもとに提案手法をツールとして実装する.

1. はじめに

C 言語や Java といったメジャーなプログラミング言語では評価戦略に先行評価を採っている．一方 Haskell は遅延評価である．遅延評価は関数呼び出しの際，関数本体の評価を行っていき，実際に引数が使われるまで引数の評価を遅らせる評価戦略である．

デバッグには，ブレークポイントの設置やステップ実行などで実行途中の変数の値を見たり，指定した関数がどのような引数で呼び出され，返り値は何かを見るといった手法がある．また実行時の評価の過程を明らかにするトレースデバッグといった手法もある．

Haskell における遅延評価に即した実行トレースを表示する場合，次のようなことが起こる．

- 未評価の式がトレース中に何度も現れる
- 関数本体を評価するトレースの中に，引数を評価するトレースが断片的に表示されるので引数と関数本体が入り組んだ表示になる

すなわち，ある式が最終的にどのような式に評価されたかという評価関係が把握しづらくなる場合がある．このことは遅延評価において，どの式の評価中にエラーが発生したのかを特定することを妨げる．また，特に先行評価であるメジャーな言語に慣れているプログラマや Haskell 初学者にとって，Haskell の式の評価に対する理解の妨げになってしまうことも考えられる．一方，先行評価の言語では引数のトレースが最終的な評価結果まで表示されたあと関数本体のトレースが表示されるので評価関係が把握しやすい．

本論文では，Haskell のトレースに対して，式の評価関係をより把握しやすくすることを目的とし，遅延評価に即したトレースを先行評価のトレースとしてに疑似的に表示する手法を提案する．また，提案手法を既存の Haskell トレースツールをもとに実装する．

2. トレース

トレースとは，プログラムの実行において計算の過程を調べることである．プログラム実行中に起こったエラーや誤りを調べるのに使われる．

トレースには，対象とする計算やその表示手法によって様々な種類がある．次に代表的なものを挙げる．

- ある関数から始まって，その関数が終了するまでの関数呼び出しの過程を表示するトレース (Common Lisp の `trace`など)
- エラーの発生した関数や式から始まって，プログラムのエントリポイントまでの関数呼び出しの過程を逆順に表示するトレース (バックトレース， スタックトレースなど)

本論文のトレースは「ある関数 (特に main 関数) から始まって，その関数が終了するまでの関数呼び出しを含めた式の評価の過程を表示するトレース」を指す．

[1] 愛知県立大学情報科学部
School of Information Science and Technology, Aichi Prefectural University
[a)] tomoki@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

$$\frac{}{\lambda \mathtt{x.\ t} \Downarrow \lambda \mathtt{x.\ t}}\ \lambda\text{ 抽象の評価}$$

$$\frac{\mathtt{t_1} \Downarrow \lambda \mathtt{x.\ t_{12}} \qquad \mathtt{t_2} \Downarrow \mathtt{v_2} \qquad [\mathtt{x} \mapsto \mathtt{v_2}]\mathtt{t_{12}} \Downarrow \mathtt{v}}{\mathtt{t_1\ t_2} \Downarrow \mathtt{v}}\ \text{適用の評価}$$

図 1 先行評価の型無し λ 計算の評価規則

2.1 λ 計算を対象とした先行評価に即したトレース

トレースの例として先行評価である型無し λ 計算のトレースについて述べる．

先行評価である型無し λ 計算の Big-Step の評価規則は**図 1** のように定義される．なお，図中の t は項，v は値，x は変数，⇓ は Big-Step での評価関係を表す．また，$[\mathtt{x} \mapsto \mathtt{v}]$ は項 t 中の変数 x を値 v に置き換えた項を表す．

2.1.1 評価過程をもとにしたトレースの表示

トレースを得るためには，トレースしたい式の評価過程を追っていく方法が考えられる．ある式 $\mathtt{t_0}$ の評価順序に即したトレースを評価過程から得るには，表示するトレースの行数を n として，**図 2** のように t の添字順に式を上から表示すればよい．

2.1.2 表示形式

2.1.1 節の方法で得られるトレースを，図 1 中の記号を使って表す．

トレース中の記号について説明する．

- 「┌○ 式」は評価する式を表す．
- 「└○ 式」は評価した結果を表す．
- 「|」は関数呼び出しの深さを表す．

「┌○ 式」と「└○ 式」は，Lisp などのトレースにおける「> 式」と「< 式」に対応する．

λ 抽象のトレースは**図 3** のように，適用のトレースは**図 4** のようになる．

2.2 Haskell を対象とした先行評価を模したトレース

2.1.1 節の方法では評価過程をもとにトレースを得るので，実際の評価を伴う必要はない．すなわち，評価過程中の式の巡回順を変えたり，疑似的な評価過程を作ることでトレースの見た目の評価規則を変えられる．

Haskell を対象とした先行評価を模したトレースとは，Haskell の遅延評価の評価過程をもとに，先行評価のように，ある関数適用のトレースを引数，関数本体の順で表示するようなトレースである．

遅延評価では「値が必要になるまで評価しない」という性質を使った式や構造が作れる．例えば，無限リストや if 式に相当する関数などがあり，これらは先行評価では表現できない．そのため Haskell での遅延評価のトレース結果を先行評価として表示すると，本来評価されない式まで評価するように表示されてしまう．

そこで先行評価を模した表現として，評価されない式の疑似的な評価結果を特別な記号「□」で表す．

2.2.1 表示形式

先行評価を模したトレースをどのように表示するか定義する．トレース中に現れる式は，関数適用，条件分岐，リテラル，変数，コンストラクタがある．ここでは特に関数適用について述べる．また変数は引数なしの関数適用として扱う．

先行評価を模したトレースにおいて関数適用は**図 5** のように表示する．これは先行評価のように引数のトレース結果をすべて表示してから，関数適用の結果を表示している．

図 5 中の各トレースは省略することがある．例えば，ある式を評価したとき結果がその式のままであれば，そのトレースは省略する．これは式が正規形 (リテラルなど) である場合が該当する．なお，トレースが省略されることはあっても，表示順序が入れ替わることはない．

次にトレース中に表示される詳細なトレースや記号について説明していく．

2.2.1.1 関数自体のトレース

関数適用の関数自体について詳細なトレースを行う．ある関数適用における関数がまた別の関数適用だった場合トレースを行う．

ここで例を示す．次のプログラムでは，既存の `zipWith`を使い，新しい `zip`関数である `zip'`を定義している．

```
1 zip' :: [a] -> [b] -> [(a, b)]
2 zip' = zipWith (,)
3
4 main = print (zip' [1, 2] [10, 20])
```

このプログラムのトレースは**図 6** のようになる．図 6 中の 2,3 行目が `zip'`について関数のトレースである．

2.2.1.2 引数のトレース

関数適用の引数のトレースを行う．関数適用に現れる引数すべてを左から順にトレースする．

ここで例を示す．次のプログラムは数値 2 つを乗算している．

```
1 main = print ((1 + 2) * (3 + 4))
```

このプログラムのトレースは**図 7** のようになる．図 7 中の 2 から 5 行目が引数のトレースである．

2.2.1.3 引数を適用した関数本体のトレース

引数を適用した関数本体のトレースの例を示す．次のプログラムでは，与えられた引数の 2 乗を返す関数 `square`を定義している．

```
1 square :: Int -> Int
2 square x = x * x
3
4 main = print (square (4 + 3))
```

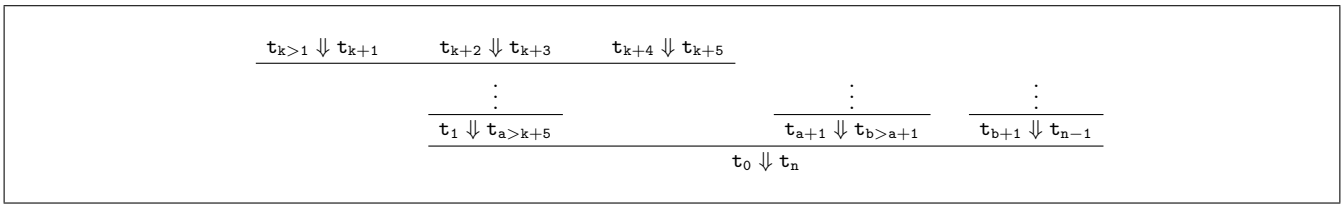

図 2 評価過程の表示順序

```
1 ┌○ λ x.t
2 └○ λ x.t
```

図 3 λ 抽象のトレース

```
1  ┌○ t1 t2
2  │ ┌○ t1 のトレース
3  │ │ ..
4  │ └○ λ x.t12
5  │ ┌○ t2 のトレース
6  │ │ ..
7  │ └○ v2
8  │ ┌○ [x |-> v2]t12 のトレース
9  │ │ ..
10 │ └○ v
11 └○ v
```

図 4 適用のトレース

```
1  ┌○ 関数適用 (= 関数 引数 0 .. 引数 n)
2  │ ┌○ 関数自体のトレース
3  │ │ ..
4  │ └○ ..
5  │ ┌○ 引数 0 のトレース
6  │ │ ..
7  │ └○ ..
8  │ ..
9  │ ┌○ 引数 n のトレース
10 │ │ ..
11 │ └○ ..
12 │ ┌○ 引数を適用した関数本体のトレース
13 │ │ ..
14 │ └○ ..
15 └○ 関数適用の結果
```

図 5 関数適用のトレース

```
1 ┌○ zip' (1 : (2 : [])) (10 : (20 : []))
2 │ ┌○ zip'
3 │ └○ zipWith (,)
4 └○ (1 , 10) : ((2 , 20) : [])
```

図 6 関数の詳細なトレースの例

```
1 ┌○ (1 + 2) * (3 + 4)
2 │ ┌○ 1 + 2
3 │ └○ 3
4 │ ┌○ 3 + 4
5 │ └○ 7
6 └○ 21
```

図 7 引数の詳細なトレースの例

このプログラムのトレースは図 8 のようになる．**図 8** 中の 4,5 行目が `square`についての引数を適用した関数本体のトレースである．

```
1 ┌○ square (4 + 3)
2 │ ┌○ 4 + 3
3 │ └○ 7
4 │ ┌○ 7 * 7
5 │ └○ 49
6 └○ 49
```

図 8 関数本体の詳細なトレースの例

```
1 ┌○ const' (1 + 2)
2 │ ┌○ 1 + 2
3 │ └○ □
4 └○ 0
```

図 9 評価されない式のトレースの例

2.2.1.4 IO アクション

IO アクションの評価結果を `IO`と表す．

2.2.1.5 条件分岐

条件分岐を表す式を「条件分岐の種類 (`if`,`case`,`|`)␣条件式」のように表示する．

2.2.1.6 評価されない式

遅延評価において，ある式がプログラムの最後まで評価されなければ，その式を先行評価で評価した擬似的な結果として未評価であることを記号「□」で表す．

ここで例を示す．次のプログラムでは，常に 0 を返す定数関数 `const'`を定義している．

```
1 const' :: a -> Int
2 const' _ = 0
3
4 main = print (const' (1 + 2))
```

このプログラムのトレースは**図 9** のようになる．

式 `1 + 2`はプログラムの最後まで評価されないので，擬似的な評価結果として「□」を表示している．

2.2.2 ラムダ式

ラムダ式を記号「λ」で表す．

ここで例を示す．次のプログラムでは，与えられた引数の 2 乗を返すラムダ式を使っている．

```
1 main = print ((\x -> x * x) (4 + 3))
```

このプログラムのトレースは**図 10** のようになる．

1 行目でラムダ式を引数に適用している．ラムダ式の実際の内容は表示せず「λ」で表している．

2.2.3 循環する式

トレースの表示において，ある式がその式自身により構

```
1 ┌○ λ (4 + 3)
2 │ ┌○ 4 + 3
3 │ └○ 7
4 │ ┌○ 7 * 7
5 │ └○ 49
6 └○ 49
```

図 10 ラムダ式のトレースの例

```
1 ┌○ take 4 ones
2 │ ┌○ ones
3 │ └○ 1 : ..
4 └○ 1 : (1 : (1 : (1 : [])))
```

図 11 循環する式のトレースの例

成されている場合がある．このような式を直接表示することはできないので，式が循環する箇所を..で表す．

ここで例を示す．次のプログラムでは，要素 1 を持つ無限リスト onesを定義している．

```
1 ones :: [Int]
2 ones = 1 : ones
3
4 main = print (take 4 ones)
```

このプログラムのトレースは**図 11** のようになる．

3 行目は実際は「1 : 1 : 1 : 1 : 1 : ...(以降無限に続く)」のような構造になっているが，このままでは表示できないので循環箇所を..と表している．

3. Hat

Hat[1] とはイギリスの York 大学で開発された Haskell プログラムをトレースするためのツール群である．執筆時点での最新バージョンは 2.9.4[2] である．Hat はプログラムのトレースを取得するツールとトレース結果を表示するツールに分かれているのが特徴である．そして，これらのツールはトレースに関する情報の入ったファイル (hat ファイル) を介し情報をやり取りする．

hat ファイルは Redex Trail という構造でトレース結果を保存している．

3.1 Redex Trail

Redex Trail とは，プログラム全体の評価を記録した有向グラフである．プログラムの実行中に現れた式をノードとして表す．そして，ある評価された式のノードは，評価元の式を親ノードとして接続する．また，評価された式が持つ部分式も，評価された式と同様の親ノードへの辺を持つ．

また，実際に使われる Redex Trail には，式と式の関係 (「式 $_1$ は式 $_2$ の引数として現れる」など) についての情報も持っている．なお Redex Trail では結果が未評価であることを表す特殊なノードが存在し，評価されなかった式の結果に，このノードが接続する．

3.2 Redex Trail の例

簡単な Redex Trail の例を示す．次のような Haskell プログラムについて考える．

```
1 f :: Int -> Int
2 f x = x + 1
3
4 main = f 2
```

このプログラムを実行しトレースすると**図 12** のような Redex Trail が得られる．

得られた Redex Trail のノードについて説明する．

- App ノードは関数適用を表す
- ""でラベルが囲まれたノードは関数やリテラルを表す

次に辺について説明する．

- parentは接続先が親であるノードを指す
- functionは接続先が適用した関数であるノードを指す
- resultは接続先が関数適用の結果であるノードを指す
- arg i は接続先が適用された i 番目の引数であるノードを指す

実際の hat ファイルには，より複雑な構造の Redex Trail が保存されている．より具体的で詳細な Redex Trail の説明は Hat に関する論文 [3][4] や hat ファイルに関するドキュメント [5] にある．[*1]

4. 先行評価を再現する巡回

Redex Trail は 2.1.1 節での評価過程に相当する．そのため，先行評価に即したトレースを得るように，Redex Trail を巡回すれば 2.2.1 節で示した表示形式が得られる．

先行評価を模したトレースを得るためのアルゴリズムを以下に記す．説明に使う関数を次に示す．

- 関数 $traverse(node)$ はノードを引数とし，文字列のリストを返す．この文字列のリストは表示するトレース結果の一行を要素として持つ．また，この関数による巡回を「深い巡回」と呼称する．
- 関数 $showExp(node)$ はノードを引数とし，文字列を返す．この文字列は表示するトレース結果の一行に相当する．また，この関数による巡回を「浅い巡回」と呼称する．
- 関数 $result(node), arg_i(node), function(node)$ はそれぞれ，引数として与えられたノードの Redex Trail 中の結果ノード, Redex Trail 中の i 番目の引数ノード, Redex Trail 中の関数ノードを返す．

***showExp*(*node*)**

引数として与えられたノードを *node* とする．

[*1] なお執筆時点の最新バージョンである，バージョン 2.9.4 とこれらの文献との間には Hat および hat ファイルの仕様にバージョンの差異がある．

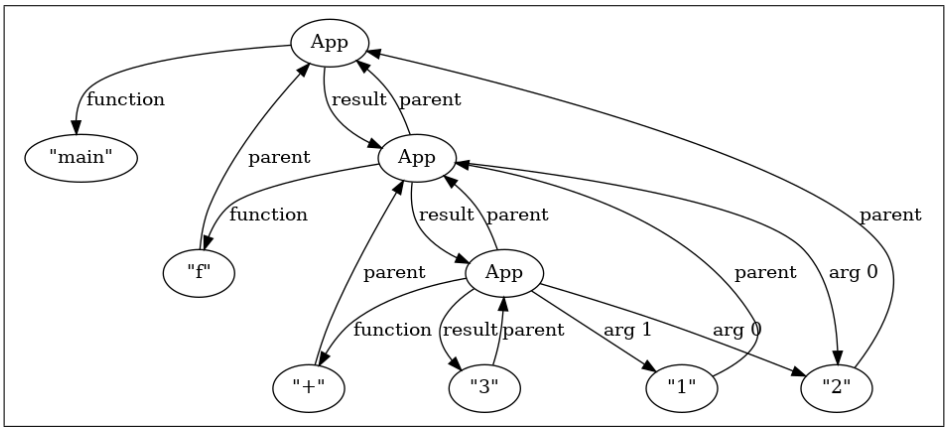

図 **12** Redex Trail の例

node が，浅い巡回について巡回済みのノード，未評価を表すノード，正規形であった場合

node に対応する文字列を返す．対応する文字列には次のようなものがある．

- リテラルであった場合，その文字列での表現
- 未評価ノードであった場合，"□"
- 浅い巡回について巡回済みであった場合，".."

node が，関数適用もしくは変数を表すノードであった場合

node に繋がってる引数ノードの数を n とする．そして，次のような順序で $n+1$ 個のリストを得る．

(1) $showExp(function(node))$ についての文字列を得る
(2) $showExp(arg_i(node))$ についての i が 0 から $n-1$ の順に n 個の文字列を得る

そして得られた $n+1$ 個の文字列の結合などを行って 1 個の文字列としたものを返す．

***traverse*(*node*)**

引数として与えられたノードを *node* とする．

node が，深い巡回について巡回済みのノード，未評価を表すノード，正規形であった場合

空リストを返す．

node が，関数適用もしくは変数を表すノードであった場合

node に繋がってる引数ノードの数を n とする．まず，$showExp(node)$ で文字列を得る．この文字列を *redexStr* とする．そして，次のような順序で文字列の $n+2$ 個のリストを得る．

(1) $traverse(function(node))$ についてのリストを得る
(2) $traverse(arg_i(node))$ について i が 0 から $n-1$ の順にリストを得る
(3) $traverse(result(node))$ についてのリストを得る

この $n+2$ 個のリストを結合したものを *subResults* とする．次に，$showExp(result(node))$ で文字列を得る．この文字列を *reducedStr* とする．そして，*node* を $result(node)$ に置き換える．最後に *subResults* の先頭に *redexStr*，末尾に *reducedStr* を追加したものを返す．

```
1 ┌○ f 4
2 │ ┌○ odd 4
3 │ └○ False
4 │ ┌○ otherwise
5 │ └○ True
6 │ ┌○ 4 - 1
7 │ └○ 3
8 └○ 3
```

図 **13** 条件式の詳細なトレースの例

そして，*traverse* 関数に `main`関数の適用を表すノードを引数として与えることで，先行評価を模したトレースを表示する．

5. 実装

4 節をもとに Haskell を使ってツールの実装を行った．ツールのソースコードは合計 1,284 行になった．

5.1 より詳細なトレース

追加の機能として，Redex Trail に存在する情報をもとにより詳細なトレースを行えるようにした．

5.2 条件式のトレース

関数適用の関数本体が `if`,`case`, ガードによって定義されている場合，その条件式に関するトレースを行う．

ここで関数定義がガードであった場合の例を示す．次のプログラムでは，与えられた引数の偶奇によって引数を加減算する関数 `f`を定義している．

```
1 f :: Int -> Int
2 f x
3   | odd x = x + 1
4   | otherwise = x - 1
5
6 main = print (f 4)
```

このプログラムのトレースは図 **13** のようになる．図 13 中の 2 から 5 行目が条件式のトレースである．

```
┌○ x + 20
│ ┌○ x
│ │ ┌○ g
│ │ │ ┌○ Just (10 + 2)
│ │ │ │ ┌○ 10 + 2
│ │ │ │ └○ 12
│ │ │ └○ Just 12
│ │ └○ Just 12
│ │ ┌○ 12
│ │ └○ 12
│ └○ 12
└○ 32
```

図 14　親ノードのトレースの例

5.3　親ノードのトレース

未巡回であった親ノードのトレースを行う．あるノードの親ノードが未巡回であるとは，評価元 (親ノード) の式が表示されないまま，評価先 (子ノード) の式を表示しようとしているのである．そのため表示しようとしているノードの親ノードが未巡回であったら，先に親ノードのトレースを表示するようにした．

ここで例を示す．次のプログラムでは，関数 gの返り値であるコンストラクタ Justの中の値を変数 xに束縛している．

```
g :: Maybe Int
g = Just (10 + 2)

main = let (Just x) = g in print (x+20)
```

このプログラムのトレースは図 **14** のようになる．図 14 中の 3 から 8 行目が親ノードのトレースである．9,10 行目がリテラル 12のトレースで，リテラル 12の親ノードのトレースが 3 から 8 行目に表示されている．説明のため 9,10 行目表示しているが，実際にはリテラルのトレース (9,10 行目) は省略する．

6.　提案手法の評価

提案手法から先行評価を模したトレースを得られるか調べる．5 節で実装したツールを使い，プログラムのトレースを行う．

6.1　テスト

自作のプログラム 30 件 (1 から 20 行程度) と Hat の example であるプログラム 7 件 (13 から 157 行程度) についてテストした．[*2]なお実行環境のメモリは 8GB であった．

Hat のトレース結果である hat ファイルが 5.1MB, 21MB であったプログラムは，トレースを得る途中でメモリが不足しテストできなかった (その他の hat ファイルは 1MB 未満であった)．残りの 35 件のプログラムについては，2.2.1 節で定義した形式，特にすべての引数のトレースが関数本体のトレースより先に表示されるという要件を満たしていた．なお，前述の 35 件のプログラムのトレース結果の総行数は 2,072 行であった．また，hat ファイルが 1.1kB のプログラムのトレースを得るのに 1.1 秒，8.1kB のプログラムは 1.3 秒，106kB のプログラムは 13.6 秒かかった．メモリが不足したプログラムは，無限ループを途中で中断したプログラムなど評価回数が多いものであった．

しかし，Prelude などの事前に用意されたライブラリ関数およびリスト内包表記について詳細なトレースが表示されなかった．また，現在の巡回アルゴリズムでは標準入力を伴うプログラムについて入力された情報が Redex Trail 内に存在するものの，その情報は表示されなかった．

6.1.1　考察

Prelude などの事前に用意されたライブラリ関数あるいは大規模なプログラムについては，先行評価を模したトレースは得られなかった．しかし小規模な自作のプログラムについては，提案手法から先行評価を模したトレースを得られた．そのため現状では初学者向けのサンプルプログラムの動作の確認などに使うことが期待される．

6.2　遅延評価との比較

次のような Haskell プログラムについて考える．

```
repl :: Int -> a -> [a]
repl 0 _ = []
repl n x = x : repl (n - 1) x

take :: Int -> [a] -> [a]
take 0 _ = []
take n (x:xs) = x : take (n - 1) xs

main = print (take 2 (repl 1 True))
```

このプログラム中の repl x nは与えられた値 xを nを個持つリストを作る関数である．また take n xsは与えられたリストの先頭から n番目までの部分リストを得る関数である．すなわち全体では，「Trueを 1 個要素として持つリストを作り，その先頭から 2 番目までの部分リストを得る」プログラムとなる．このプログラムを実行すると Error: No match in pattern.というエラーになる．

このプログラムが，どのような原因でエラーを起こしているかをトレース結果から特定する場合を考える．エラーを起こす関数の候補は repl， takeである．実際には takeの定義に，nが 0 になる前にリストが空になった場合を記述しておらず，7 行目のリストのパターンマッチに失敗するため起こる．

まず遅延評価に即したトレースを図 15 に示す．そして先行評価を模したトレースを図 16 に示す．

遅延評価でのトレースでは，20 行目で発生したエラーが 9 行目の takeで発生しているとわかる．そして takeの引数である，14 行目での replが 19 行目で空リストになってい

[*2] Hat の example は，Hat のソースコード [2] で参照できる．

```
1  ┌○ take 2 (repl 1 True)
2  │ ┌○ 2 == 0
3  │ └○ False
4  │ ┌○ repl 1 True
5  │ │ ┌○ 1 == 0
6  │ │ └○ False
7  │ └○ True : (repl (1 - 1) True)
8  │ ┌○ True : (take (2 - 1) (repl (1 - 1) True))
9  │ │ ┌○ take (2 - 1) (repl (1 - 1) True)
10 │ │ │ ┌○ (2 - 1) == 0
11 │ │ │ │ ┌○ 2 - 1
12 │ │ │ │ └○ 1
13 │ │ │ └○ False
14 │ │ │ ┌○ repl (1 - 1) True
15 │ │ │ │ ┌○ (1 - 1) == 0
16 │ │ │ │ │ ┌○ 1 - 1
17 │ │ │ │ │ └○ 0
18 │ │ │ │ └○ True
19 │ │ │ └○  []
20 │ │ └○ Error: No match in pattern.
21 │ └○ True : Error: No match in pattern.
22 └○ True : Error: No match in pattern.
```

図 15 遅延評価に即したトレース

```
1  ┌○ take 2 (repl 1 True)
2  │ ┌○ repl 1 True
3  │ │ ┌○ 1 == 0
4  │ │ └○ False
5  │ │ ┌○ True : (repl (1 - 1) True)
6  │ │ │ ┌○ repl (1 - 1) True
7  │ │ │ │ ┌○ 1 - 1
8  │ │ │ │ └○ 0
9  │ │ │ │ ┌○ 0 == 0
10 │ │ │ │ └○ True
11 │ │ │ └○ []
12 │ │ └○ True : []
13 │ └○ True : []
14 │ ┌○ 2 == 0
15 │ └○ False
16 │ ┌○ True : (take (2 - 1) [])
17 │ │ ┌○ take (2 - 1) []
18 │ │ │ ┌○ 2 - 1
19 │ │ │ └○ 1
20 │ │ └○ Error: No match in pattern.
21 │ └○ True : Error: No match in pattern.
22 └○ True : Error: No match in pattern.
```

図 16 先行評価を模したトレース

るので，関数 `repl`ではエラーが発生していないとわかる．そして 9 行目の `take`で空リストから長さ 1 の部分リストを取り出そうしエラーになったとわかる．

一方，先行評価でのトレースでは 2 行目の `repl`が 13 行目 `True : []`に評価された，すなわち関数 `repl`ではエラーは発生していないとわかる．そして 16 行目の `take`で空リストから長さ 1 の部分リストを取り出そうし，その結果である 20 行目がエラーになっているとわかる．

6.2.1 考察

先行評価のトレースでは `repl`の評価と `take`の評価が分かれて表示されていた．このことで，先行評価でのトレースでは，より早い段階で `repl`でエラーが発生していないとわかり，`take`の使い方が誤っているとわかった．そのため，どの関数で，どのような引数が渡されてエラーが発生したかが，よりわかりやすくなったと考えられる．

7. おわりに

7.1 まとめ

本論文では，Haskell における遅延評価に即したトレースに比べ，先行評価を模したトレースの方が式の評価関係が把握しやすいという仮説に基づき，その実現手法を提案した．また，Haskell におけるトレースツールである Hat を使い，提案手法をツールとして実装した．そして，提案手法から先行評価を模したトレースを得られるか，遅延評価よりも式の評価関係をより把握しやすくなったことを確認した．

7.2 今後の課題

7.2.1 アルゴリズムや実装の改善

現状の巡回アルゴリズムでは，標準入力を伴うプログラムについて入力された情報が Redex Trail 内に存在するものの，その情報は表示されない．さらに，現状の実装では，Hat の仕様上 Prelude などの事前に用意されたライブラリ関数の詳細なトレースが表示されない．上記の問題のためアルゴリズムや実装の改善が必要であると考えられる．

7.2.2 部分的なトレース機能の追加

現状の実装では，大規模なプログラムのトレースを行うとメモリが不足する問題がある．しかし実用上では大規模なプログラム全体をトレースするのではなく，その一部分についてトレースすることが多いと考えられる．そこでプログラムの指定した範囲のみをトレースする機能を追加することで，メモリが不足する問題を回避できると考えられる．

謝辞 本研究は JSPS 科研費 15K00488 の助成を受けたものである．

参考文献

[1] The haskell tracer hat. `https://archives.haskell.org/projects.haskell.org/hat`.
[2] Github - olafchitil/hat: The haskell tracer. `https://github.com/OlafChitil/hat`.
[3] Thorsten Brehm Malcolm Wallace, Olaf Chitil and Colin Runciman. Multiple-view tracing for haskell: a new hat. *Preliminary Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pp. 151–170, 2001.
[4] Colin Runciman Chitil, Olaf and Malcolm Wallace. Transforming haskell for tracing. *Symposium on Implementation and Application of Functional Languages*, pp. 165–181, 2002.
[5] The definition of the augmented redex trail. `https://www.cs.york.ac.uk/fp/hat/Memos/augmentedRedexTrail.2.ps.gz`.