修士論文

Haskellの広範な関数に対応したメモ化による高速化手法

指導教員　松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学大学院工学研究科
修士課程創成シミュレーション工学専攻
平成 21 年度入学　21413517 番

大野 将臣

平成 23 年 2 月 3 日

Haskellの広範な関数に対応したメモ化による高速化手法

大野 将臣

内容梗概

計算機の高性能化に伴い，実行可能なプログラムの規模が大きくなってきたことや，求められる機能が多様化してきたことにより，プログラムの開発コストが増加してきた．そのため，より生産性の高いプログラミング言語が求められるようになってきており，柔軟なプログラミングが可能で，かつ高い可読性を持つ関数型言語に注目が集まってきている．しかし，関数型言語はC言語などの手続き型言語に比べ動作が低速であり，その動作速度の向上が望まれている．

一方で，これまでに様々なプログラム高速化手法が研究されてきた．その高速化手法の一つにメモ化が存在する．メモ化とは過去の計算結果を記憶しておくことで，再び同じ計算が現れたときに，その記憶した計算結果を再利用することで計算を省略し高速化する手法である．

そこで，本論文では単一代入という特徴を持つ関数型言語Haskellで記述されたプログラムにメモ化を適用するとき，関数の入出力を記憶した入出力表を複数の関数呼び出し間で共有するのに制約が存在することを明らかにする．そして，この制約による制限を受けずにHaskellプログラムにメモ化を自動的に適用する二つの手法を提案する．その手法の一つとして，Haskellプログラムを自動的に変換し，複数の関数呼び出しの間で関数の入出力を記憶した入出力表を共有することで，関数の実行結果を再利用する手法を提案する．さらに，プログラムにメモ化を適用した際に問題となる，入出力表の検索オーバヘッドを削減するための手法を提案し，プログラムの更なる高速化を図る．

また，Haskellコンパイラを拡張し，プログラムのコンパイル過程でその中間コードおよびランタイムシステムにメモ化を実現するためのコードを自動挿入することで高速化を図る手法の提案と，その実現方法の検討を行う．

提案手法の有効性を検証するために，Haskellプログラムに対し本提案手法およびその高速化手法を適用した時のプログラム実行時間を計測し評価を行った．その結果，再帰関数に提案手法を適用した場合，既存手法と比べ最大で約2.03倍の高速化を実現した．また，非再帰関数に提案手法を適用した場合，通常実行時と比べ最大で約1.81倍の高速化を実現し，その有効性を示した．

Haskellの広範な関数に対応したメモ化による高速化手法

目次

1 はじめに 1

2 研究背景 2

2.1 関数型言語 2

2.2 メモ化 2

2.3 Haskell 4

2.3.1 Haskellの記述方法 5

2.3.2 Haskellの特徴 7

3 Haskellプログラムへのメモ化の適用 9

3.1 Monadic Memoization Mixins 9

3.2 既存手法の問題点 12

3.3 Haskellプログラムにメモ化を適用する際の制約 14

4 Haskellプログラムの自動変換によるメモ化 15

4.1 ラッパー関数 15

4.2 プログラムの自動変換 18

4.2.1 変換必要関数の判定 18

4.2.2 呼び出し元関数の変換 20

4.3 入出力表検索オーバヘッドの削減 22

4.3.1 外部関数呼び出しによる入出力表検索の高速化 22

4.3.2 検索と関数実行の並行化による検索オーバヘッドの隠蔽 25

5 コンパイラ拡張によるメモ化手法の検討 28

5.1 GHCの中間コード 29

5.2 メモ化コードの挿入個所 30

5.3 引数の取得 32

5.4 入出力表 36

5.4.1 入出力表領域の確保 36

5.4.2 入出力表への値の登録 38

5.4.3 入出力表の検索 39

6 評価 40

6.1 評価環境 40

6.2 実行時間の比較 42

7 おわりに 48

7.1 まとめ 48

7.2 今後の課題 49

謝辞 50

参考文献 50

1　はじめに

計算機の高性能化に伴い，実行可能なプログラムの規模が拡大してきたことによって，プログラムの開発コストが増大してきた．そこで，より生産性の高いプログラミング言語が求められるようになってきたことから，効率的にプログラムを記述することができる関数型言語に注目が集まってきている．関数型言語は可読性が高く，高階関数や型推論など効率的にプログラムを記述することができる機能を有しているにも関わらず，手続き型言語に比べ広く普及しているとは言い難い．これは，手続き型言語で記述されたプログラムに比べ，関数型言語で記述されたプログラムは動作速度が遅く実用に耐えなかったためである．そのため関数型言語処理系の高速化が求められている．

一方で，プログラムの高速化手法として様々なものが提案されてきた．プログラム中の並列性に着目し，命令レベルやスレッドレベルでプログラムを並行実行することで高速化する手法があるが，これらの並列性に着目した手法とは異なるプログラム高速化手法にメモ化がある．

メモ化とは値の局所性に着目した高速化手法で，過去の計算結果を記憶しておくことで，再び同じ計算が現れたときに，その記憶した計算結果を再利用することで計算を省略し高速化する手法である．一般に，プログラム中でメモ化を適用できる関数は，その関数に対する入力以外の要素に出力結果が依存しない関数である．関数型言語で記述されたプログラムでは，引数以外の要素に依存する関数と依存しない関数を容易に区別することができる．そのため，関数型言語は自動的にメモ化を適用しやすい言語であると言える．

そこで本論文では，関数型言語の一つである Haskell で記述されたプログラムに自動的にメモ化を適用することで高速化を図る二つの手法を提案する．一つ目の手法は，Haskell プログラム中のメモ化対象関数呼び出しの間で，関数の入出力を記憶する入出力表を共有可能なように変換することでメモ化を実現する手法である．また，二つ目の手法は，Haskell コンパイラを拡張し，コンパイルされたコードおよびそのコードを実行するランタイムシステムにメモ化コードを挿入することでメモ化を実現する手法である．

以下，本論文では，2 章で本研究の背景と本提案手法の対象となる関数型言語 Haskell について述べる．3 章で，Haskell プログラムにメモ化を適用する既存研究とその問題点ついて述べ，Haskell プログラムにメモ化を適用する際の制約を明らかにする．続い

て，4 章で Haskell プログラムを自動的に変換しメモ化を適用する提案手法について述べ，5 章では Haskell コンパイラを拡張することによってメモ化を適用する提案手法について述べる．6 章では，本提案手法の評価とそれに対する考察を述べ，最後に 7 章で本論文全体をまとめる．

2　研究背景

2.1　関数型言語

計算機の高性能化に伴い，実行可能なプログラムの規模が大きくなってきた．また，携帯電話や自動車といった様々な組み込み機器に求められる機能が増加してきたことにより，多様な機器への対応や様々な機能を実現するために，プログラムの開発コストが大きくなってきている．このようなプログラム規模の拡大や開発コストの増大により，より生産性の高いプログラミング言語が求められるようになってきたことから，C 言語などの手続き型言語に比べより柔軟なプログラミングを行うことができる関数型言語に注目が集まってきている．

関数型言語は手続き型言語に比べ可読性が高く，柔軟で効率のよいプログラミングを行うことができる [1]．たとえば，変数や関数の引数および戻り値の型を自動的に決定する型推論により，プログラマが誤った型を使用するのを回避することができ，バグの発生を抑制することができる．また，多くの関数型言語は関数の評価戦略に遅延評価を採用しており，遅延評価される関数型言語では，関数の出力値が必要でない限り関数の実行は後回しにされる．このため，ある関数を実行するときその関数が必要とする値を生成する式だけが計算され，必要とされない値は計算されず，無駄な計算を遅延させることで計算量を低減させることができる．

このように，関数型言語は効率的なプログラムを実現する遅延評価などの機能を有していることや可読性の高いプログラムを記述することができることなどから，生産性の高いプログラミング言語として注目が集まっている．しかし，関数型言語は C 言語などの手続き型言語に比べ，動作速度が遅く高速化が求められている [2]．

2.2　メモ化

これまでに，プログラムの高速化手法として，命令レベルでプログラムを並列化する手法が研究されてきた．命令レベルの並列性に着目した手法として，SIMD（Single Instruction Multiple Data）が存在する．SIMD とは，一命令で複数のデータに対し同一の処理することで高速化を実現する手法である．しかし，SIMD を用いることがで

```
1  int x = 1;
2
3  int powX(int a){
4    return (pow(x, a));
5  }
6
7  int main (){
8    pow(2, 3);
9    pow(2, 3);
10
11   powX(2);
12   x = 2
13   powX(2);
14   return 0;
15 }
```

入出力表			
関数	入力		出力
pow	a = 2	b = 3	8
powX	x = 1	a = 2	1
powX	x = 2	a = 2	4

図 1: 関数へのメモ化の適用

きるのは画像処理など複数のデータに対して同じ演算を繰り返す操作などの場合に限られる．

一方，CPU のマルチコア化に伴い，より粒度の荒い並列性であるスレッドレベル並列性が注目されている．例えば，その手法として並列化ライブラリを用いたり，自動並列化コンパイラを用いてコンパイル時に自動的に複数のコアに処理を割り当てる手法がある．しかし，並列化ライブラリを用いた場合，プログラマが明示的に並列化可能な処理をスレッドの形で記述する必要があり，プログラマの負担が増加してしまう．また，自動並列化コンパイラはその精度に問題があり，逐次プログラムの完全な並列化を行うことはできない．

さて，プログラムの並列化とは全く別の高速化手法としてメモ化がある．メモ化とは，関数などの命令区間の入力と出力を記憶しておき，後で再び同じ入力でその計算が行われようとしたときに，過去に記憶した計算結果を再利用することで，再計算を省く手法である．メモ化により一度行った計算を省略することができ高速化を図ることができる．たとえば，**図 1** に示すプログラムの巾乗を計算する関数 `pow` にメモ化を適用したとき，8 行目で初めて関数 `pow` が呼び出されると，その入力である 2，3 とその出力である 8 が図 1 の右に示す関数とその入力および出力を記憶する表である入出

力表へ記憶される．そして，9 行目でメモ化を適用した関数 pow が 8 行目の呼び出し時と同じ引数 2，3 で呼び出されたとき図 1 の入出力表に記憶された出力 8 を再利用する．このように入出力表に記憶された計算結果を再利用することで，計算を省略し高速化を行うことができる．

しかし，プログラム中の関数にメモ化を適用したい場合，メモ化可能な関数は参照透過性を備えたものに限られる．参照透過性を備えた関数とは，その関数に同じ入力を与えたとき，常に同じ値を返す関数である．つまり，メモ化可能な関数はファイル入出力などの入力以外の状態に依存する処理が関数の中に含まれていない関数である．また，図 1 の 3-4 行目に示す関数 powX のように関数内で大域変数などの引数以外の入力を参照している関数にメモ化を適用する場合，関数中で参照している全ての入力を確認する必要がある．たとえば， 11 行目と 13 行目で関数 powX は全く同じ引数 2 で呼び出されているが，関数 powX 中で参照している大域変数 x の値が 12 行目で変更されているため，これらの計算結果は異なる．このため，入出力表に関数 powX の入力を記憶する際には，関数中で参照されている大域変数 x も入力として記憶する必要がある．このように， C 言語のなどの手続き型言語によって記述されたプログラムにメモ化を適用したとき，関数に明示的に与えられる引数以外の入力も確認しなければならない．そのため，メモ化の適用を自動的に行おうとしたとき，関数の入力の依存関係の解析が複雑化してしまう．

一方，関数型言語には，状態という概念が存在しないため，明示的に与えられた引数以外に依存する関数はほとんどない．このため，関数にメモ化を適用したとき，ほとんどの関数はその引数だけを確認することでよい．しかし，関数の実行結果が引数以外に依存する場合がある．それは例えば，プログラム実行時にコマンドラインから引数を受け取ったり，実行結果を印字するなどの入出力処理に相当する．このような場合，関数型言語では，入出力結果などの引数以外の状態が入出力処理を含まない関数に影響を及ぼさないようにするための機構を用いて，これらの関数を区別している．そのため，関数型言語はメモ化可能な参照透過性を備えた関数を容易に判別でき，メモ化を自動的に適用しやすい言語であると言える．

2.3 Haskell

関数型言語の一つに Haskell[3][4][5] がある．本節では，既存手法および提案手法を説明する際に必要となる，Haskell プログラムの記述方法と Haskell の特徴について述べる．

```
1  -- 関数名 :: 第1引数の型 -> 戻り値の型
2  -- 関数名 仮引数 = 関数本体の式
3  square :: Int -> Int
4  square x = x ^ 2
5
6  -- 関数名 :: 第1引数の型 -> 第2引数の型 -> 戻り値の型
7  -- 関数名 仮引数1 仮引数2 = 関数本体の式
8  quotRem :: Int -> Int -> (Int, Int)
9  quotRem x y = (div x y, mod x y)
10
11 (quot, rem) = quotRem 4 3
12
13 cube :: Num a => a -> a -> a
14 cube x = x ^ 3
15
16 pow6 = cube . square
17
18 fib :: Int -> Int
19 fib 0       = 0
20 fib 1       = 1
21 fib (n + 2) = fib n + fib (n + 1)
22
23 type FilePath = String
```

図 2: Haskell の記述方法

2.3.1 Haskell の記述方法

Haskell で関数を定義する場合，**図 2** に示すように,「::」の右にその関数の引数と戻り値の型を記述する．関数の引数の型と戻り値の型は「->」で区切って記述する．さらに，関数名のあとにその関数の仮引数と「=」を記述し，その右に関数で行う処理を記述する．たとえば， 3 行目に示す，引数の平方を返す関数 square の例では， Int 型の引数を一つとり，Int 型の値を返すことを示しており， square は引数を仮引数 x として受取り，その平方の値を返す．

また，関数が二つ以上の引数を取るような場合も同様に,「::」の右に「->」を用いて引数および戻り値の型を区切って記述する．左から順に第 1 引数の型，第 2 引数の型

と記述し，最後に戻り値の型を記述する．関数が 2 つ以上の値を返すような場合，それらの値を，複数の要素の組を表すデータ構造であるタプルにして返す．Haskell では，タプルは各要素をコンマで区切り，全体を括弧で括って表現する．たとえば，8-9 行目に示す商と余りを求める関数 quotRem は Int 型の二つの引数を受取り，商と余りを求め，その結果である二つの値をタプルにすることで返す．

関数呼び出しには，関数名のあとにその関数の引数を記述することで引数を渡すことができる．たとえば図 2 の 11 行目に示すように，関数 quotRem に引数 43 を適用すると，その結果である商と余りをそれぞれ quot と rem として受け取ることができる．

また，Haskell には型推論機能があるため，関数の引数や戻り値の型を必ずしも記述する必要はなく，図 2 内の関数の引数や戻り値を記述した行を省略することができる．ただし，型推論により関数の引数および戻り値の型を記述する必要が無い場合でも，型を明記することでプログラムの可読性を向上させることができるため，本論文のプログラム例では型を記述している．

Haskell では型の定義を抽象化することが可能で，図 2 の 13 行目の関数 cube に示すように，任意の型を示す型名 a を用いてその関数の引数および戻り値の型を定義することができる． a は抽象化された型であるため，その関数はすべての型の値を引数に取ることができるが，三乗を求める関数 cube の場合，引数に取れる型は Int 型や Float 型などの数値型に限られる．そのような場合,「=>」指示子を用いて， 13 行目に示すように，Num a =>と記述することで a の型に条件をつけることができる．この例では， Num a により型 a は数値型であることを示している．

さらに，Haskell では複数の関数から一つの新しい関数を作り出すことができ，これを関数合成と呼んでいる．たとえば，ある値の六乗を計算する関数を定義したいとき， 図 2 の square や cube と同様に x^6 と定義することもできるが，関数合成を用いて square と cube から六乗を計算する関数を定義することができる．そこで，関数を合成するための演算子である「•」を用いて cube • square と二つの関数を合成し，新たな関数 pow6 として定義する．関数合成演算子「•」によって h = f • g と二つの関数 f，g を合成した場合， h は引数 x を与えると f(g(x)) のように g，f の順に適用される関数として定義される．例えば図 2 の 16 行目に示すように，square，cube の順に適用される新しい関数 pow6 を定義することができる．

また，Haskell では，関数に渡される引数の値に応じて，関数本体の定義を別々に記述することができ，if 文などを使って記述した場合に比べ可読性の高い記述を実現できる．例えば図 2 の 18 行目のフィボナッチ数を求める関数 fib は，引数が 0 のとき 0

```
1 main = do let a = 1 + 2
2           let a = 3 + 4  -- 束縛不可能
3           let b = 3 + 4
```

図 3: 単一代入

を返し，引数が 1 のとき 1 を返し，それ以外のときは引数を (n + 2) として受け取り，fib が再帰的に呼ばれるように定義されている．入力された引数が複数の条件に当てはまるような場合には，当てはまる定義の内の一番上の定義が採用される．

また，Haskell では type 宣言を用いることで型に別名をつけることが可能であり，可読性を向上させることができる．たとえば，図 2 に示すように，type 宣言を用い，String 型に FilePath と別名を与えることにより，新たに FilePath 型を使うことができる．

2.3.2 Haskell の特徴

Haskell で記述されたプログラムにメモ化を適用する際に問題となり得る Haskell の特徴として，単一代入と IO モナドがある．

単一代入 Haskell には，変数に値を割り当て直す演算である代入を行う式は存在せず，変数は最初に定義された値と常に同じである．例えば，**図 3** に示すプログラムの場合，1 行目で局所変数を導入する式である let により，変数 a が a = 1 + 2 と定義されている．同一スコープ内では変数 a は常に 1 + 2 という値をとる．このとき，仮に 2 行目に示すように let a = 3 + 4 とすると，プログラムコンパイル時にエラーとなってしまう．

このように，Haskell では一度定義された変数に対して再び値を定義し直すことはできない．そのため，2 行目の 3 + 4 を変数として参照するには変数 a とは全く別の変数として定義する必要がある．たとえば，3 行目に示すように変数 a とは別の変数 b として定義する．つまり，Haskell では,「=」演算子は代入を行うための演算子ではなく，値や式に変数名を与えるための演算子である．たとえば 1 行目の a = 1 + 2 は，変数 a に 1 + 2 の計算結果を代入しているのではなく，1 + 2 という式に対して変数名 a を与えている．このように，値や式に変数名を与えることを束縛という．

IO モナド Haskell では，入出力などの副作用を伴う操作を含まない純粋関数と，副作用を伴う操作を含む非純粋関数がある．純粋関数とは，引数以外の状態に戻り値が依存せず，同じ引数を与えたときに常に同じ値を返す関数である．このため，純粋関

```
1 square :: Int -> Int
2
3 x = square 10
4
5 readFile :: FilePath -> IO String
6
7 y <- readFile "filename"
```

図 4: 純粋関数と非純粋関数

数は実行順に依存せず，どのような順で実行した場合でも実行結果は常に同じになる．これに対し非純粋関数とは，関数の引数以外の状態によって動作が変化する可能性のある関数や，他の関数の動作に影響を及ぼす可能性のある関数である．たとえば，ファイル読み出しを行うような関数の場合，ファイルに書き込まれているデータによって，その関数が返す結果が変わってしまう．また，ファイルへ出力を行う関数の場合，その関数の戻り値以外の状態に変化を与えるため，ファイル読み出し関数の動作が変わってしまう可能性がある．

Haskell では，このような純粋関数と非純粋関数を IO モナドという機構を用いて明確に区別している．たとえば，**図** 4 の 1 行目に示す純粋関数 `square` の戻り値の型は `Int` 型となっているのに対し，5 行目のファイルから文字列を読み出す非純粋関数 `readFile` の戻り値の型は `IO String` 型となっている． このように Haskell では非純粋関数の戻り値に IO タグが付加されるため，純粋関数か非純粋関数かどうか容易に判別することができる．

このように IO タグが付加された変数から実際の値を取り出すには,「`<-`」演算子を用いる．たとえば，図 4 では，`readFile` によって返される `IO String` 型の値から，実際に読み出した `String` 型の文字列を「`<-`」演算子によって用いて取りだし，変数 y にその文字列を束縛している．

さらに，Haskell では非純粋関数が及ぼす影響が純粋関数にまで波及しないように，IO タグの付加された非純粋関数は同様に IO の付加された非純粋関数内からしか呼び出すことはできない．たとえば，図 4 に示す，純粋関数である `square` から非純粋関数である `readFile` を呼び出すことはできない．しかし，その逆の非純粋関数から純粋関数を呼ぶことは可能である．そのため，関数の引数以外の状態に依存する動作を非純粋関数に閉じ込めることができ，入出力などの副作用を伴う処理に起因するバグの特

```
gFib :: (Int -> Int) -> (Int -> Int)
gFib self 0       = 0
gFib self 1       = 1
gFib self (n + 2) = self n + self (n + 1)

retFunc 0       = 0
retFunc 1       = 1
retFunc (n + 2) = func n + func (n + 1)

fibg :: Int -> Int
fibg = fix gFib

type Gen a = a -> a

gmFib :: Monad m => Gen (Int -> m Int)
gmFib self 0       = return 0
gmFib self 1       = return 1
gmFib self (n + 2) = do a <- self n
                        b <- self (n + 1)
                        return (a + b)
```

図 5: 関数 fib の変換

定を容易にすることができる．

3 Haskell プログラムへのメモ化の適用

Haskell で記述されたプログラムにメモ化を適用する手法として，Monadic Memoization Mixins[6] が提案されている．本章ではこの手法の問題点を述べ，Haskell プログラムにメモ化を適用する際の制約を明確にする．

3.1 Monadic Memoization Mixins

Haskell で記述されたプログラムにメモ化を適用する手法として Monadic Memoization Mixins が提案されている．

図 2 で示した，フィボナッチ数を求める関数である fib にこの手法を適用する場合を説明する．はじめに，関数 fib を**図 5** の 1 行目に示すような関数 gFib に書き換える．

gFib は引数に関数をとり，戻り値として関数を返す関数である． gFib の引数である関数は図 5 の 1 行目に (Int -> Int) とあるように，Int 型の引数を受け取り，Int 型の値を返す関数である．Haskell では関数の引数や戻り値に Int 型の値などと同じように関数を取ることができる．戻り値も同様に (Int -> Int) によって示される関数である．この gFib に (Int -> Int) 型の関数を与えると，gFib はその関数を self として受け取る．gFib の仮引数 self のあとの引数は，gFib が返す関数の引数である．たとえば，gFib に関数 func を引数として与えたとすると， gFib の返す関数は図 5 の 6 行目の retFunc のような関数で， retFunc は fib 内の再帰呼び出しだった個所が func に置き換えられた関数である．関数 fib を gFib のように変換し，その引数に関数を渡すことで， fib 内では再帰呼び出しだった個所を任意の関数に置き換えることが可能になる．

この gFib を fix の引数として渡した関数を fibg とする（10-11 行目）．fix 関数は最小不動点を返す関数である．すなわち， fix 関数に関数 f を与えたとき，f(x) = x となる x を返す．この fix 関数に gFib を与えると，gFib(func) = func となるような関数 func が返される．fix によって返された関数 func，つまり fibg は図 2 の fib と同じ動作をする関数である．このとき，gFib のように関数を引数にとり関数を返す関数を fix の引数に与えると新たな関数を得ることができる．この gFib のような関数の型を type 宣言を用い 13 行目に示すように Gen a 型と別名をつける．この Gen a 型を用いると関数 gFib を gFib :: Gen (Int -> Int) と記述することができる．

さて，関数にメモ化を適用する場合，その関数の入力と出力を記憶する表が必要になり，その入出力表をメモ化対象関数間で共有する必要がある．この既存手法では，再起関数内の再帰呼び出しを置き換えた関数の間で入出力表を共有するために State モナドと呼ばれる機構を用いている．この例では，gFib でモナドを利用できるように，図 5 の 15 行目に示すように gFib を gmFib 関数に変換している．gmFib 関数の Monad m により m が汎用的なモナド型を表す Monad 型であることを示し， type 宣言により導入された Gen によって引数と戻り値の型を表現している．gmFib 関数の引数は (Int -> m Int) によって示される関数で，戻り値も同様の関数である．モナドを用いた関数はその戻り値を return によって返すため，gmFib 関数も同様に return によって戻り値を返すように変換される．

メモ化を実現する場合，関数の入出力が記憶された入出力表を検索する検索関数や，関数の入出力を表へ登録する登録関数が必要になる．そこで，入出力表の検索と値の登録を行う関数の型を type 宣言を用い，**図 6** の 1 行目に示すような Dict a b m 型

```
type Dict a b m = (a -> m (Maybe b),
                   a -> b -> m ())

memo :: Monad m => Dict a b m -> Gen (a -> m b)
memo (check, store) super x = do
  y <- check x
  case y of
    Just z  -> return z
    Nothing -> do z <- super x
                  store x z
                  return z

type Memoized a b m = Dict a b m -> a -> m b
memoFib :: Monad m => Memoized Int Int m
memoFib dict = fix (memo dict . gmFib)

mapDict :: Ord a => Dict a b (State (Map a b))
mapDict = (check, store) where
  check a   = gets (lookup a)
  store a b = modify (insert a b)

memoMapFib :: Int -> State (Map Int Int) Int
memoMapFib = memoFib mapDict

runMemoMapFib :: Int -> Int
runMemoMapFib n = evalState (memoMapFib n) empty
```

図 6: メモ化関数の適用

と別名をつける．Dict a b m 型は検索関数と登録関数のタプルである．検索関数は任意の型 a を引数にとり m (Maybe b) を返す関数である． Maybe b は Just b または Nothing を値として取る型で，検索が成功したときの検索結果を Just b と表現し，検索が失敗したときの検索結果を Nothing と表現している．登録関数は任意の型 a，b の二つを引数にとり空の値を返す関数である．そして，検索関数と登録関数を呼び出しメモ化を行う関数を図 6 の 4 行目に示す memo 関数として定義する．memo 関数は入出力表の検索と値の登録を行う関数を (check,store) のようにタプルとして受け取り，

メモ化対象となる関数を super として受け取り，そのメモ化対象関数 super の引数を x として受け取っている．はじめに，memo 関数は入出力表の検索を行いその結果を y に束縛する．検索結果は値が見つからなかった場合には，y == Just z になり， z にはメモ化対象関数の出力値が束縛される．値が見つからなかった場合には，検索結果は y == Nothing になり，本来実行されるはずであった関数 super x を実行し，その結果を入出力表へ登録する．

そして，図 6 の 14 行目の memoFib 関数に示すように， memo 関数と gmFib 関数を関数合成演算子を用いて (memo dict • gmFib) のように合成する．このとき，dict は入出力表の検索や入出力表へ値の登録を行う関数のタプルである．そして，この合成した関数に対して fix を適用することで，gmFib 内の self にメモ化を適用した関数を束縛することができ，メモ化を実現することができる．

入出力表の検索と入出力表への値の登録を行う関数のタプルを図 6 の 17 行目の mapDict として定義する．mapDict は入出力表の検索を行う check 関数と，入出力表へ値の登録を行う store 関数のタプルである．さらに，この手法ではメモ化対象関数の再帰呼び出しの間での入出力表の受け渡しを隠蔽するために，State モナドを用いている．通常，gmFib の self 呼び出しの間で入出力表を共有するには， self の引数に入出力表を記述する必要がある．しかし，State モナドを用いることで，入出力表の受け渡しをプログラマから隠蔽することができ，self の引数に入出力表を明示的に記述する必要がなくなる．入出力表には連想配列である Map を利用している． check 関数は State モナドから Map を取りだし検索を行い， store 関数は State モナド内の Map に値を登録している．

そして，memoFib に対し，mapDict として定義した入出力表に対する操作を引数として渡した関数を図 6 の 22 行目の memoMapFib として定義する．この memoMapFib に対し，図 6 の 25 行目の runMemoMapFib に示すように，空の入出力表を渡し，State モナドを用いた関数を実行する関数である evalState に渡すことで fib にメモ化を適用することができる．既存手法は再帰関数に対して，このような書き換えを行うことでメモ化を実現している．

3.2 既存手法の問題点

既存手法は再帰関数を書き換え，再帰関数内の再帰呼び出しをメモ化適用済の関数に置き換え，それらの関数の間で入出力表を共有することでメモ化を実現している．そのため，既存手法は再帰呼び出しの内部でしかメモ化を適用することができない． 例

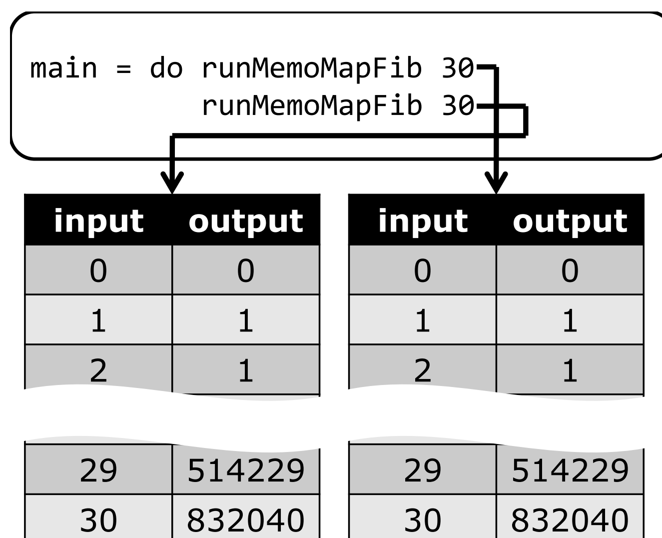


図 7: 再利用不可能な例

えば**図 7** で示すようなプログラムを実行するとき，`runMemoMapFib` を全く同じ引数で呼び出しているにも関わらず，一度目の呼び出しの時に入出力表へ記憶した入出力結果を，二度目の `runMemoMapFib` 呼び出しの時に再利用することができず，一度行った計算を再び実行してしまうことになる．このように，再帰関数であっても図 7 のような，複数のメモ化対象関数に跨った再利用はできない．

さらに，既存手法では，メモ化を適用したい関数を 3.1 節で示した変換手順に従いプログラマが書き換える必要がある．そのために，プログラマは変換方法を理解する必要がある． 3.1 節で示した変換例である `fib` 関数は引数に `Int` 型の値を一つ取り，戻り値が `Int` 型であったが，引数を二つ以上取るような関数に対してこの手法を適用したい場合や，戻り値の型がモナドとなっているような場合，その手順は複雑となる．

例えば**図 8** の 1 行目に示す関数 `unfringe` にこの手法を適用したい場合を考える．`unfringe` 関数はリストを受け取り，木構造のリストデータに変換し返す関数である．`unfringe` は受け取ったリストを分割し，その分割したリストを引数に `unfringe` を再帰的に呼ぶ．このような関数を既存手法に従い， 3.1 節で関数 `fib` から関数 `gmFib` に変換したように，`unfringe` を変換すると図 8 の `gmUnfringe` のようになる（9-21 行目）．このように，Tree モナドなどのモナドを用いた関数を書き換える場合には，図 5 のような単純な書き換えだけでは，この既存手法を適用することはできない．分割したリストを 16，17 行目の `self` に引数として渡すことで， `m [Tree a]` 型の値を取得することができ，さらにここで得られた値から本来取得したい値である `[Tree a]` 型の値

```
unfringe :: [a] -> [Tree a]
unfringe [a] = [Leaf a]
unfringe as  = do
  (l, k) <- partitions as
  t      <- unfring l
  u      <- unfring k
  return (Fork t u)

gmUnfringe :: Monad m => Gen ([a] -> m [Tree a])
gmUnfringe self [a] = return [Leaf a]
gmUnfringe self as  =
  liftM concat (
    sequence (do
                (l, k) <- partitions as
                return (do
                           ts <- self l
                           us <- self k
                           return (do
                                      t <- ts
                                      u <- us
                                      return (Fork t u)))))
```

図 8: モナドを返す関数の変換例

を取得するために 19，20 行目で<-演算子を用いている．

このように，メモ化を適用したい関数の引数や戻り値の型によって変換する方法が異なる．そのため，プログラマがこの手法をプログラムに適用したいと考えたとき，大きな負担となってしまう．

3.3 Haskell プログラムにメモ化を適用する際の制約

関数にメモ化を適用する際に，その関数の入力と出力を記憶する表が必要になる．既存手法では，その入出力表を再帰関数内の再帰呼び出し間でのみ共有していた．そのため，メモ化を適用することができる関数が再帰関数に限定されてしまっていた．これは，Haskell の特徴の一つである単一代入からくる制約によるものである．

Haskell では単一代入により，一度定義された変数へ再び値を代入することはでき

ず，値を変数として定義するには全く別の変数として定義する必要がある．このため，Haskell プログラムにメモ化を適用する場合，関数の入出力を記憶する入出力表を変数として定義し，その変数に対し新たに入出力を登録した入出力表を代入することはできず，入出力を登録した新たな入出力表を別の変数として定義し直すことになる．

また，関数のメモ化を実現するには，入出力表をメモ化対象関数呼び出しの間で共有し，過去の計算結果を再利用する必要がある． Haskell では，入出力表をグローバル変数として定義することができないため，メモ化対象関数の間で入出力表を共有したい場合，メモ化対象関数の引数として入出力表を受け取り，そして更新された入出力表を戻り値として返す必要がある．このため，メモ化を適用したい関数の入力と戻り値の型を変更する必要がある．

一方， 2.3.2 項で述べたの IO モナドを用いることで，一度定義された変数に対して再び値を代入することができる．そのため，IO モナドを用いて入出力表を実現した場合，C 言語などのグローバル変数と同様に，どの関数からも参照し，更新することが可能になる．しかし，この入出力表を参照，更新する関数も IO モナドを利用した非純粋関数となってしまい，メモ化を適用した関数を純粋関数から呼び出すことができなくなってしまう．このため，IO モナドを用いて入出力表を実現した場合，IO モナドを利用した非純粋関数にしかメモ化を適用することができない．そこで，次章でこれらの制約に対し，複数の関数間で入出力表を共有可能なようにプログラムを自動変換することでメモ化を実現する手法を提案する．

4　Haskell プログラムの自動変換によるメモ化

Haskell プログラムを自動的に変換し，再帰関数に限らない全ての関数にメモ化を適用することができる手法を提案する．この手法では，プログラマがメモ化を適用したい関数を指定するだけで，自動的にプログラムを変換し，指定された関数にメモ化を適用する．

4.1　ラッパー関数

関数にメモ化を適用する場合，その関数の入出力を記憶した表を検索する操作や，入出力表に新たに入出力を登録する操作が必要になる．そこで，この提案手法ではメモ化対象関数ごとに，入出力表の検索や入出力の登録を行うラッパー関数を作成する．このラッパー関数を呼び出すことで，入出力表に記憶された過去の計算結果を再利用することができる．

```
1  square :: Int -> Int
2  square x = x * x
3
4  type Table = Map Int Int
5  square_wrap :: Int -> Table -> (Int, Table)
6  square_wrap arg table = (ret, newtable)
7    where
8      search_result = lookup arg table       -- 入出力表の検索
9      ret           = case search_result of
10       Just r  -> r                    -- 検索が成功したとき
11       Nothing -> square arg           -- 検索が失敗したとき
12     newtable = if (isJust search_result)
13                  then table
14                  else insert arg ret table -- 値の登録
```

図 9: ラッパー関数

たとえば，**図 9** に示す，関数 square（1-2 行目）にメモ化を適用したい場合，関数 square のラッパー関数 square_wrap（5-14 行目）を定義する．このラッパー関数 square_wrap は，メモ化を適用したい関数 square の引数の Int 型の値に加え，新たに Table 型の入出力表を引数にとる．Table 型は type 宣言により新たに定義した型で，キーが Int 型の値でデータが Int 型の値である連想配列として定義されている．また，ラッパー関数 square_wrap の戻り値は，メモ化対象関数 square が返す値の Int 型の値と新たに入出力の登録された入出力表のタプルである．ラッパー関数 square_wrap は，メモ化対象関数 square が引数として受け取る値を arg として受け取り，過去の実行結果が記憶された入出力表を table として受け取っている．

はじめに，ラッパー関数は入出力表の検索を行う関数 lookup により，arg をキー値として入出力表 table を検索し，その結果を search_result に束縛する（8 行目）．次に，case 文によって検索結果 search_result の値に応じて分岐し，ラッパー関数が返す ret に適切な値を束縛する（9-11 行目）．検索が成功し値が見つかった場合には，search_result の値は Just r になり，r には検索により見つかった値が束縛される（10 行目）．検索が失敗し値が見つからなかった場合には，search_result の値は Nothing になり，square arg を呼び出す（11 行目）．この結果，ret には r または square arg が束縛される．そして，ラッパー関数は入出力表の更新を行い，新たな入

```
1  main = let (val1, tb1) = square_wrap 10 empty
2             (val2, tb2) = square_wrap 20 tb1
3             (val3, tb3) = square_wrap 10 tb2
```

empty

tb1	
10	100

tb2	
10	100
20	400

図 10: ラッパー関数の呼び出し

出力表を newtable として定義する（12-14 行目）．検索が成功し値が見つかった場合，入出力表にはすでに入出力が登録されているため，新たな入出力表を作成する必要がない．そのため，新たな入出力表 newtable に table を束縛する（13 行目）．検索が失敗し値が見つからなかった場合，引数と square の実行結果を登録する必要がある．そのため，既存の入出力表に新たな入出力を加えた入出力表を返す関数である insert 関数によってキー値を arg，データを ret として追加した入出力表を newtable に束縛する（14 行目）．そして，ラッパー関数は本来の戻り値 ret と新たな入出力表 newtable のタプルを戻り値として返す（6 行目）．

このラッパー関数が複数回呼び出されるとき，その呼び出しの間で更新された入出力表を共有する必要がある．**図 10** に示すように，ラッパー関数が main 文から複数回呼び出される場合を考える．はじめに 1 行目に示すようにラッパー関数 square_wrap が 10 と空の入出力表 empty を引数として呼ばれたとき，square_wrap は入出力表 empty の検索を行うが，入出力表は空のため，square が引数を 10 として呼ばれる．そして，その結果が入出力表に登録され，square_wrap は square の返す値と更新された新たな入出力表をタプルとして返す． square の返す値と新たな入出力表をそれぞれ val1 と tb1 に束縛する（1 行目）． main 文では，再びラッパー関数 square_wrap が呼ばれるときに，1 行目の呼び出し時に入出力の登録された入出力表 tb1 を引数に渡すことで，square_wrap の間で入出力表を共有することができる．2 行目でラッパー関数 square_wrap が 20 と tb1 を引数として呼び出されたとき，入出力表にキー値が 20 の入出力は存在しないため，先ほどの square_wrap 呼び出し時と同様に，入出力が入出力表に登録され，square の戻り値と新たな入出力表がそれぞれ，val2 と tb2 に束縛される．さらに 3 行目で square_wrap が 10 と tb2 を引数として呼び出されたとき，2 行目で更新された入出力表 tb2 には入力が 10 の入出力が存在するため，その出力である 100 を関数 square を呼び出すことなく再利用することができる．

```
1 #MEMOIZATION iofunction
2 iofunction :: Int -> IO Int
3 iofunction  a = ...
```

図 11: 参照透過性の保証

一方，IO モナドを用いた関数にメモ化を適用する場合，その関数は参照透過性を備えている必要がある． 2.3.2 目で述べたように，IO モナドを用いた関数は，その関数内部で入出力操作を行っている可能性があるため，単純にメモ化を適用することはできない．そのため，その IO モナドを用いた関数が参照透過性を備えていることをプログラマによって保証してもらう必要がある．そこで，IO モナドを用いた関数にメモ化を適用したい場合には，#MEMOIZATION プラグマを用いてメモ化対象関数が参照透過性を備えていることを保証する．たとえば，**図 11** に示す，関数 iofunction は IO モナドを用いた関数である．この関数にメモ化を適用するには，1 行目に示すように，#MEMOIZATION によって関数 iofunction が参照透過性を備えていることを保証する必要がある．

提案手法ではラッパー関数の引数と戻り値を介することで，複数の関数呼び出しの間で入出力表の共有を可能にし，関数が再帰関数で無い場合にもメモ化を適用することができる．また，既存手法を適用するにはプログラマによる関数の書き換えが必要であったが，提案手法ではこのプログラム変換を自動的に行う．自動変換手法を次節以降で説明する．

4.2 プログラムの自動変換

4.2.1 変換必要関数の判定

メモ化対象となっている関数が様々な関数から呼ばれている場合， 4.1 節で説明したラッパー関数を作成し，メモ化対象関数の呼び出しをラッパー関数の呼び出しに置き換えるだけでは，ラッパー関数間で入出力表を共有することはできない．入出力表をラッパー関数間で共有するには，メモ化対象関数の呼び出し元関数の変換も必要になる．

たとえば，**図 12** の左に示すようなプログラムの関数 square にメモ化を適用したい場合を考える．このとき関数の呼び出し関係は図 12 の右に示すようになっており，main から関数 A が呼び出され，関数 A から関数 B と C が呼び出され，関数 B と C から

```
 1  A :: Int -> Int
 2  A x = B x + C x
 3
 4  B :: Int -> Int
 5  B x = 2 * (square x)
 6
 7  C :: Int -> Int
 8  C x = square x
 9
10  main = print (A 30)
```

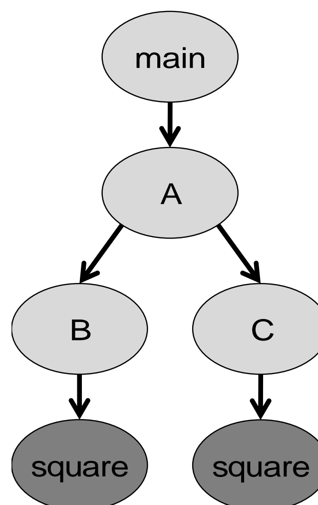


図 12: メモ化対象関数の呼び出し関係

```
1  fib :: Int -> Int
2  fib 0     = 0
3  fib 1     = 1
4  fib (n+2) = fib n + fib (n+1)
5
6  X :: Int -> Int
7  X n = fib n
```

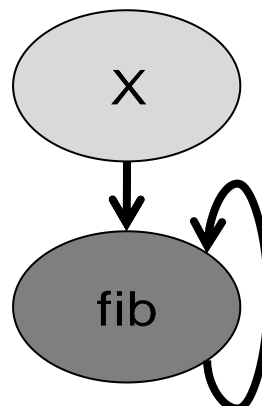


図 13: 再帰関数の呼び出し関係

メモ化対象である関数 square が呼ばれている．

図 12 の右に示される木構造のリーフにあたる関数 square 間で入出力表を共有するには，メモ化対象関数 square から呼び出し元を遡っていき，すべてが交わるまでの関数を変換することで，メモ化対象関数 square の間で入出力表を共有することができる．図 12 の場合では，関数 A までの関数を変換する必要がある．つまり，関数 A，関数 B および関数 C の全てを変換する必要がある．

また，メモ化対象関数が**図 13** の左に示す再帰関数 fib の場合， 図 13 の右に示すように，再帰関数 fib は関数 X と自分自身から呼び出されることになる．そのため，再帰関数がメモ化対象の場合はメモ化対象関数自身も変換する必要がある．

```
 1  B_t :: Int -> Table -> (Int, Table)
 2  B_t x in_tb = (ret, out_tb)
 3    where (val1, tb1) = square_wrap x in_tb
 4          ret         = 2 * val1
 5          out_tb      = tb1
 6
 7  C_t :: Int -> Table -> (Int, Table)
 8  C_t x in_tb = (ret, out_tb)
 9    where (val1, tb1) = square_wrap x in_tb
10          ret         = val1
11          out_tb      = tb1
12
13  A_t :: Int -> Int
14  A_t x = ret
15    where (val1, tb1) = B_t x empty
16          (val2, tb2) = C_t x tb1
17          ret         = val1 + val2
```

図 14: 呼び出し元の変換

4.2.2 呼び出し元関数の変換

メモ化対象関数の呼び出し元関数を変換するとき，図 12 の関数 A のようにメモ化対象関数から遡って行ったときにすべてが交わる位置に存在する関数の場合と，関数 B や C のように呼び出し関係を遡って行くときの途中に存在する関数の場合とでは変換方法が異なる．

図 12 の関数 B，C の場合は，**図 14** の 1-5 行目に示す関数 B_t および 7-11 行目の関数 C_t のように変換される．関数 B_t は変換前関数の引数 x に加え入出力表を新たに引数 in_tb として受け取り，戻り値が変換前関数の戻り値と入出力表のタプルになるように変換された関数である（1 行目）． 図 12 の変換前の関数 B でメモ化対象関数 square の呼び出しであった個所がラッパー関数 square_wrap の呼び出しへと変換され，ラッパー関数の square_wrap の戻り値であるタプルの各要素が val1 と tb1 に束縛される（3 行目）．そして，変換前の関数が返す値であった (2 * square x) の square x はラッパー関数の戻り値の val1 に置き換えられる（4 行目）．さらに，ラッパー関数 square_wrap によって更新された入出力表 tb1 は out_tb に束縛され（5 行目），ラッパー関数 square_wrap の返す値は戻り値 ret と更新された入出力表 out_tb のタプル

に変換される（2 行目）．関数 C に対しても同様の変換を行う．このように，引数として入出力表を受け取り，戻り値として更新した入出力表を返すことで，呼び出し元関数とこの関数が呼び出している関数の間で入出力表を受け渡ししている．

一方，図 12 の関数 A の変換例を図 14 の 13-17 行目に示す．関数 A はメモ化対象関数のすべての呼び出し元関数であるため，入出力表を引数として受け取ったり，戻り値として返す必要がない．そのため，関数 A の引数と戻り値を変換する必要はない．しかし，関数 A から呼び出される関数 B および C の間で入出力表を受け渡しするために，関数 B と C の呼び出しを変換する必要がある．関数 B の呼び出しを図 14 の 15 行目の B_t のように変換する．B_t の呼び出しより前にメモ化対象関数が一度も呼ばれていないため，関数 B_t の引数に空の入出力表 empty を渡す．そして，関数 B_t の戻り値をそれぞれ val1 と tb1 に束縛する．関数 B_t から呼び出されるラッパー関数 square_wrap によって更新された入出力表を tb1 に束縛しているため，その入出力表 tb1 を関数 C_t の引数に与えることで，関数 B_t で行った計算結果を関数 C_t から呼び出されるラッパー関数 square_wrap で再利用することができる（16 行目）．

また，メモ化対象関数が図 13 の関数 fib のように再帰関数の場合，メモ化対象関数自身も変換する必要がある．このとき関数 fib は**図 15** の 1-8 行目に示す関数 fib_t に変換される．関数 fib はメモ化対象関数を呼び出す関数であるため，図 14 の関数 B_t と同様に，新たに入出力表を引数に取り，戻り値が関数の実行結果と更新した入出力表のタプルとなる関数に変換される．引数が 0 または 1 のとき（2-3 行目），メモ化対象関数の呼び出しが無いため，入出力表の更新は行われず，受け取った入出力表 in_tb をそのまま返す．引数が 0，1 でない場合，関数 fib の再帰呼び出しはラッパー関数 fib_wrap の呼び出しへと変換される（5-6 行目）．さらに，ラッパー関数 fib_wrap 中のメモ化対象関数呼び出しが関数 fib_t の呼び出しへ変換され，その戻り値のタプルの各要素が val1 と tb1 に束縛される（16 行目）．そして，入出力表を更新するとき（19 行目），tb1 にキー値を arg データを ret として fib の入出力を登録する．

メモ化対象として指定された関数に対し，このような変換を自動的に行うことで複数の関数から呼ばれるメモ化対象関数の間で入出力表を共有することができる．そのため，メモ化対象関数が再帰関数でない場合でも入出力表を共有することができ高速化を実現できる．

```
 1 fib_t :: Int -> Table -> (Int, Table)
 2 fib_t 0       in_tb = (0, in_tb)
 3 fib_t 1       in_tb = (1, in_tb)
 4 fib_t (n + 2) in_tb = (ret, out_tb)
 5   where (val1, tb1) = fib_wrap n in_tb
 6         (val2, tb2) = fib_wrap (n + 1) tb1
 7         ret         = val1 + val2
 8         out_tb      = tb2
 9
10 fib_wrap :: Int -> Map Int Int -> (Int, Table)
11 fib_wrap arg table = (ret, newtable)
12   where search_result = lookup arg table
13         ret = case search_result of
14           Just r -> r
15           Nothing -> val1
16         (val1, tb1) = fib_t arg table
17         newtable = if (isJust search_result)
18                      then table
19                      else (insert arg ret tb1)
20
21 X_t :: Int -> Table -> (Int, Table)
22 X_t n in_tb = (ret, out_tb)
23   where (val1, tb1) = fib_wrap n in_tb
24         ret         = val1
25         out_tb      = tb1
```

図 15: 再帰関数の変換

4.3 入出力表検索オーバヘッドの削減

関数にメモ化を適用したとき，メモ化を適用する以前には無かった入出力表を検索するオーバヘッドが発生する．そこで，本提案手法では，入出力表を検索するオーバヘッドを削減することで更なる高速化を図る．

4.3.1 外部関数呼び出しによる入出力表検索の高速化

C 言語などの手続き型言語は，Haskell などの関数型言語に比べ動作速度が速い．そこで，関数にメモ化を適用したときに，オーバヘッドとなる入出力表の検索操作および入出力表への値の登録操作を C 言語で記述した関数を生成し，これらの検索関数お

```
int table[SIZE] = { 0 };

int lookup(int x){
  return table[x];
}

void update(int x, int y){
  table[x] = y;
}
```

図 16: C 言語で記述した入出力表操作関数

よび値の登録関数を Haskell プログラムから呼び出すことで，入出力表に対する操作を高速化しオーバヘッドを削減する．

Haskell には，他の言語で書かれたコードを呼び出したり，あるいは他の言語で書かれたコードから Haskell のコードを呼び出すためのインターフェースとして Foreign Function Interface（FFI）[7] が用意されている．この FFI を用いて，C 言語で記述された関数を呼び出す．

はじめに，C 言語で記述された入出力表および入出力表の検索関数，値の登録関数を生成する必要がある．たとえば，図 9 に示す関数 `square` にメモ化を適用する場合を考える．**図 16** に示すように，入出力表，およびその入出力表に対する検索と値の登録を行う関数を生成する．関数 `square` の引数および戻り値の型は共に `Int` 型であるため，入出力表を図 16 では `int` 型の配列で表現している（1 行目）．そして，入出力表の検索関数 `lookup` は引数に `int` 型の値 x を受け取り，入出力表の x 番目の要素を返す関数である（3-5 行目）．また，入出力表へ値を登録する関数 `update` は `int` 型の値 x,y を受け取り，入出力表の x 番目に y を代入する関数である（7-9 行目）．これらの入出力表，その入出力表の検索関数，および値の登録関数は，メモ化対象関数の引数や戻り値の型によって，変更する必要がある．

Haskell プログラム側では，FFI を用い C 言語で記述した関数をインポートし，ラッパー関数内の入出力表検索関数と入出力表への値の登録関数を置き換える．関数 `square` にメモ化を適用する場合，**図 17** に示すようになる．はじめに，図 16 で示した C 言語によって記述された検索関数 `lookup` と値の登録関数 `update` を FFI を用いてインポートする． C 言語で記述された関数をインポートするには `foreign` 宣言を用いて図 17

```
1  foreign import ccall "lookup" c_lookup :: CInt -> IO CInt
2  foreign import ccall "update" c_update :: CInt -> CInt -> IO ()
3
4  square_wrap :: Int -> IO Int
5  square_wrap arg = do
6    search_result <- c_lookup arg        -- 入出力表の検索
7    ret <- case search_result of
8      0 -> do let val = square arg       -- 検索が失敗したとき
9              c_update (hashInt arg) val -- 入出力の登録
10             return val
11     b -> return b                      -- 検索が成功したとき
12   return ret
```

図 17: 外部関数呼び出し

の 1-2 行目に示すように記述し，検索関数 lookup を c_lookup に束縛し，値の登録関数 update を c_update に束縛する．そして c_lookup 関数と c_update 関数の引数および戻り値の型を「::」の右に記述する．

C 言語で記述された検索関数 lookup の引数および戻り値の型は共に int 型であるため，c_lookup 関数の引数と戻り値の型も同様に Int 型となる（1 行目）．しかし，C 言語で記述された lookup 関数内で入出力表である配列にアクセスし，その入出力表の状態によって引数が同じであっても戻り値が異なることがある．そのため，c_lookup 関数は IO モナドを用いた非純粋関数としてインポートする必要がある．また，値の登録関数 update の引数は int 型の二つの値で戻り値は void であるため， c_update 関数の引数に Int 型の値を二つ取り，引数は空の値を表す () となる（2 行目）．さらに，update 関数内では入出力表である配列に値の書き込みを行っているため，検索関数と同様に IO モナドを用いた非純粋関数としてインポートする必要があり，戻り値が IO () となる．

ラッパー関数は FFI によってインポートした関数を利用することで，入出力表の操作を高速化しオーバヘッドを削減する． 図 9 に示される square_wrap の検索関数と入出力表を更新する関数を C 言語で記述した関数に置き換えると図 17 の square_wrap のようになる． 図 9 では square_wrap の引数および戻り値を介して，入出力表を呼び出し元関数と受け渡ししていた．しかし，この高速化手法では入出力表も検索関数や値の登録関数と同様に C 言語で定義されているため，square_wrap の引数や戻り

値を介して呼び出し元関数と受け渡しする必要がなくなる．ただし，IO モナドを用いた非純粋関数を呼び出すことのできる関数は同様に IO モナドを用いた関数であるため，square_wrap も IO モナドを用いた関数として定義する必要がある．そのため，square_wrap の戻り値の型が IO Int となる．

図 17 の square_wrap では，はじめにインポートした c_lookup 関数により入出力表を検索し，その結果を search_result に束縛している（6 行目）．戻り値 ret の値を case 文により検索が失敗したときと，成功したときで分岐し決定する．検索が失敗し検索結果 search_result が 0 のとき，square_wrap はメモ化対象関数 square を呼び出し，その戻り値を val に束縛し（8 行目），インポートした値の登録関数によりその値を入出力表に登録し（9 行目），その値を返す（10 行目）．val を登録する際，hashInt 関数を用いて引数 arg のハッシュ値をキー値として利用している．このため，このハッシュ関数の定義されている型（Int 型，String 型）以外の値を引数に取る関数にこの手法を適用する場合，プログラマにハッシュ関数を定義してもらう必要がある．また，検索が成功し検索結果 search_result の値が 0 以外の場合，その値を b に束縛しその値を返す（11 行目）．戻り値 ret は検索が失敗した場合には val に束縛され，検索が成功した場合には b に束縛される．そして，square_wrap は戻り値として ret を返す．

このように，ラッパー関数内の検索関数および値の登録関数をより実行速度の高速な C 言語で記述した関数へ置き換えることで，オーバヘッドを削減し高速化することができる．しかし，C 言語で記述した関数と Haskell プログラムの間で受け渡しすることができるデータの型は FFI の仕様により限定されているため，任意の関数に対してこの高速化手法が適用できるわけではない．利用することが可能なデータの型を**表 1** に示す．また，表 1 に示した型のデータを受け渡しする場合であっても，それらのデータがリストやタプルなど C 言語の型で直接表現できない場合やメモ化を適用したい関数がモナドを利用した関数の場合はこの手法を適用することはできない．

4.3.2 検索と関数実行の並行化による検索オーバヘッドの隠蔽

関数にメモ化を適用した場合，ラッパー関数は入出力表の検索を行い，入出力表に検索対象が無かったとき，メモ化対象関数の実行を行う．このとき，入出力表の検索と関数の実行を並行して行うことができれば，入出力表検索が失敗したときの検索オーバヘッドを隠蔽することができる．

そこで，**図 18** に示すように，入出力表の検索と関数の並行実行が可能なようにラッパー関数を拡張する手法を提案する．はじめにラッパー関数は newEmptyMVar により空の同期変数を作成し m に束縛する．同期変数はスレッド間で通信を行うための変数

表 1: FFI で利用できる型

C 言語での型名	Haskell での型名
char	CChar
short	CShort
int	CInt
long	CLong
long long	CLLong
float	CFloat
double	CDouble

で，複数のスレッドでこの変数を共有し，値を同期変数を介して受け渡しすることでスレッド間の通信を行う．

ラッパー関数はスレッドの作成を行う関数 `forkIO` を用いて，検索スレッドと関数実行スレッドの二つのスレッドを作成し，各スレッドのスレッド ID を `t1` と `t2` に束縛する．スレッド作成関数 `forkIO` は引数として受け取った関数を実行するスレッドを作成し，関数の実行が終わるとそのスレッドは破棄される．検索スレッド `t1` では，`c_lookup` 関数により入出力表の検索を行い，その結果を `search_result` に束縛する．検索が成功し検索結果 `search_result` の値が 0 以外の値の場合，その値を `tryPutMVar` により同期変数 `m` に格納しようと試みる．すでに同期変数 `m` に値が格納されている場合には，何も行わずに終了する．検索に失敗し検索結果 `search_result` の値が 0 の場合，空の値を示す `()` を返し，関数の実行が終了しスレッドが破棄される．関数実行スレッド `t2` では検索スレッドの検索結果に関わらず関数 `square` の実行を行う．そしてその結果を `val` に束縛し，その値を `c_update` により入出力表に登録する．次に，検索スレッド同様にその値を `tryPutMVar` により同期変数 `m` に格納し，実行を終了する．

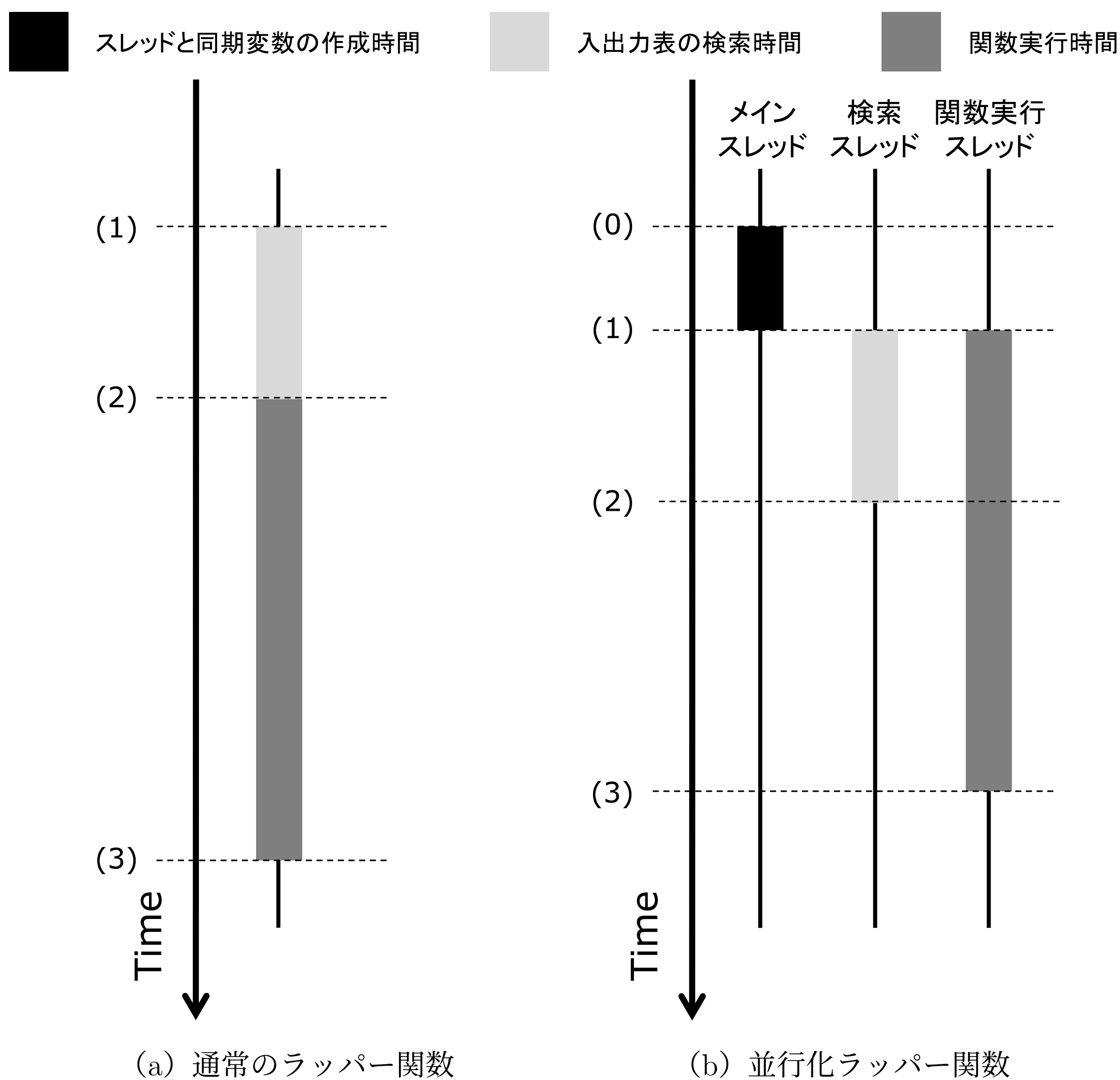
ラッパー関数は二つのスレッド，検索スレッドおよび関数実行スレッドを作成した後，`takeMVar` 関数により同期変数 `m` に値が格納されるまで待機する． `takeMVar` 関数は引数に与えられた同期変数に値が格納されるまでスレッドを待機させ，同期変数に値が格納されるとその値を取得する関数である．ラッパー関数は同期変数 `m` に値が格納されるとその値を取得し `ret` に束縛する．同期変数に値が格納されたとき，検索スレッドまたは関数実行スレッドによって戻り値としてラッパー関数が返す値が得られている．そのため，両スレッドをこのまま実行させたままにしておくことは無駄であ

```
square_wrap :: CInt -> IO CInt
square_wrap arg = do
  m <- newEmptyMVar -- 同期変数の作成
  -- 検索スレッドの作成
  t1 <- forkIO (
    do search_result <- c_lookup arg    -- 入出力表の検索
       if (search_result /= 0)
         then tryPutMVar m search_result -- 同期変数へ値の登録
         else return ()
    )
  -- 関数実行スレッドの作成
  t2 <- forkIO (
    do let val = square n -- 関数の実行
       c_update n val     -- 入出力表へ値の登録
       tryPutMVar m val   -- 同期変数へ値の格納
       return ()
    )
  ret <- takeMVar m -- 同期変数から値の取得
  killThread t1 -- 検索スレッドの破棄
  killThread t2 -- 関数実行スレッドの破棄
  return ret
```

図 18: 検索と関数実行の並行実行

るため，両スレッドを killThread 関数によって破棄する．killThread 関数によって検索スレッドおよび関数実行スレッドの破棄を行わなかった場合でも，両スレッドは実行が終了すると自動的に破棄される．しかし，メモ化対象関数が再帰関数であった場合，関数実行スレッドでは再帰呼び出しによって，再びラッパー関数が呼び出され，スレッドの作成が行われてしまい，スレッドが大量に生成される可能性がある．そのため，ラッパー関数の最後にスレッドの破棄を行っている．

入出力表の検索が失敗し値が見つからないとき，入出力表の検索と関数実行の並行化を適用していないラッパー関数は，**図 19** の（a）に示すように，(1）入出力表の検索を行った後に，(2）関数の実行を行う．一方，入出力表の検索と関数実行の並行化を適用したラッパー関数は，図 19 の（b）に示すように，(0）スレッドと同期変数の作成を行い，（1）入出力表の検索とメモ化対象関数の実行を並行に行う．そのため，



(a) 通常のラッパー関数　　(b) 並行化ラッパー関数

図 19: 通常のラッパー関数との比較

スレッドと同期変数の作成に掛かる時間が，入出力表の検索に掛かる時間と比べ小さいとき入出力表の検索オーバヘッドを隠蔽することができ，並行化を適用しなかったラッパー関数に対し，実行時間を短縮することができる．

5　コンパイラ拡張によるメモ化手法の検討

Haskell コンパイラを拡張し，Haskell プログラムのコンパイル過程で自動的にメモ化コードを挿入し，メモ化を実現する手法検討する．

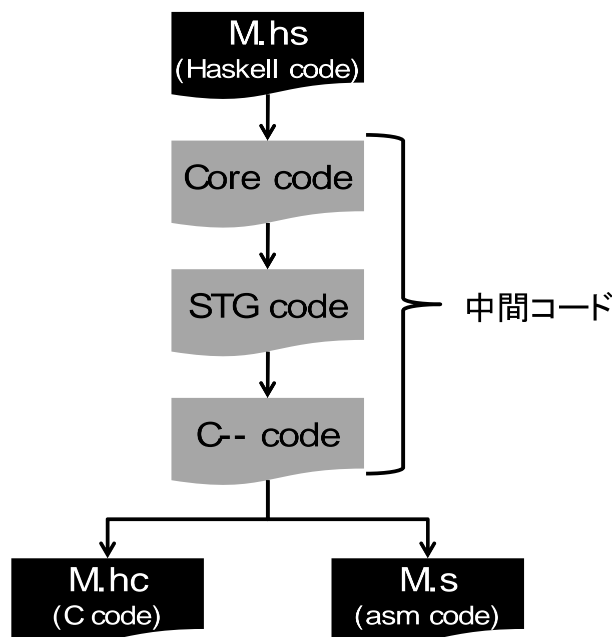


図 20: GHC のプログラムコンパイル手順

5.1 GHC の中間コード

Haskell コンパイラの一つである Glasgow Haskell Compiler（GHC）[8] を拡張し，Haskell プログラムのコンパイル過程でメモ化コードを自動挿入することで，関数のメモ化を実現する手法を検討する．GHC は，最も広く知られた Haskell コンパイラの一つで，事実上標準の Haskell コンパイラであるため，本提案手法の対象コンパイラとした．

GHC のプログラムコンパイル手順は**図 20** に示すように， Haskell コードから三つの中間コード，Core コード，STG コード， C--コード [9] に変換された後，実行プログラムへと変換される． C--コードは C 言語に似たプログラミング言語で，人間が読み書きを行うための高級言語ではなく，コンパイラなどで生成・解析されることを目的とした言語である．実行プログラムは**図 21** に示すような構成となっている． GHC によって Haskell コードから変換された C--コードは仮想的なマシンのマシンコード列で，この C--コードが仮想的なマシンである GHC Runtime System （RTS）によって解釈・実行される．

RTS はスタック領域やヒープ領域といったメモリ領域の管理やガベージコレクションを行う Storage Manager， スレッドの発行やスケジューリングを行う Scheduler，関数の実行時間やヒープ領域の利用状況を調査する Profiler から構成されている． RTS

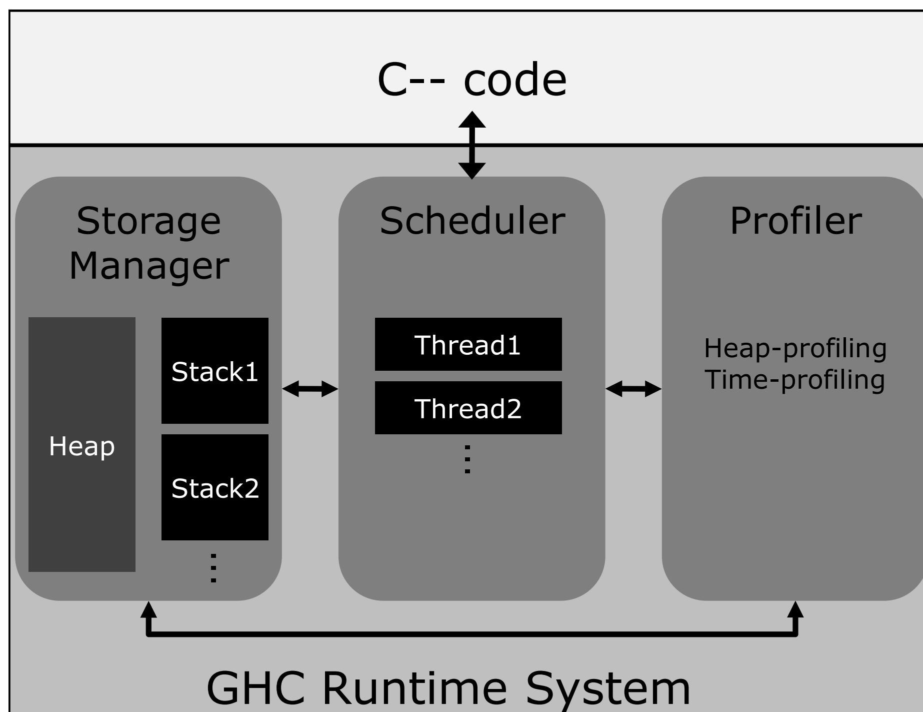


図 21: 実行プログラム

の Scheduler によって発行されたスレッドごとにスタック領域が確保され，スレッドが C--コードを解釈・実行していくことでプログラムの実行が行われる．

5.2　メモ化コードの挿入個所

本提案では，GHC によってコンパイルされたコードである C--コードへのメモ化コードの挿入および，そのコードを解釈・実行する RTS を拡張することでメモ化を実現する．**図 22** の左上に示す Haskell プログラムを GHC でコンパイルする図 22 の右に示す C--コードへと変換される．この Haskell コード中の関数および引数の評価順は，図 22 の左下の図のようになっており，`main` から `print` の引数の `(fib 35)` が評価され，そこから `fib` が呼び出され，そして `print` が呼ばれている． C--コードでは，Haskell プログラム中の関数および引数にあたる部分が関数として定義されており，この例ではそれぞれ `main`，`sat`，`fib`，`print` がそれにあたる．また，Haskell プログラムの関数 `fib` の引数 35 が C--コードでは，変数 `x` として `x = 35` と定義されてる． C--コードでは関数の呼び出しを `jump` によって行う． 図 22 の C--コードでは，`main` 関数から `jump` により関数 `sat` が呼び出され，同様に関数 `fib` が呼び出され，さらに関数 `print` が呼び出されている．

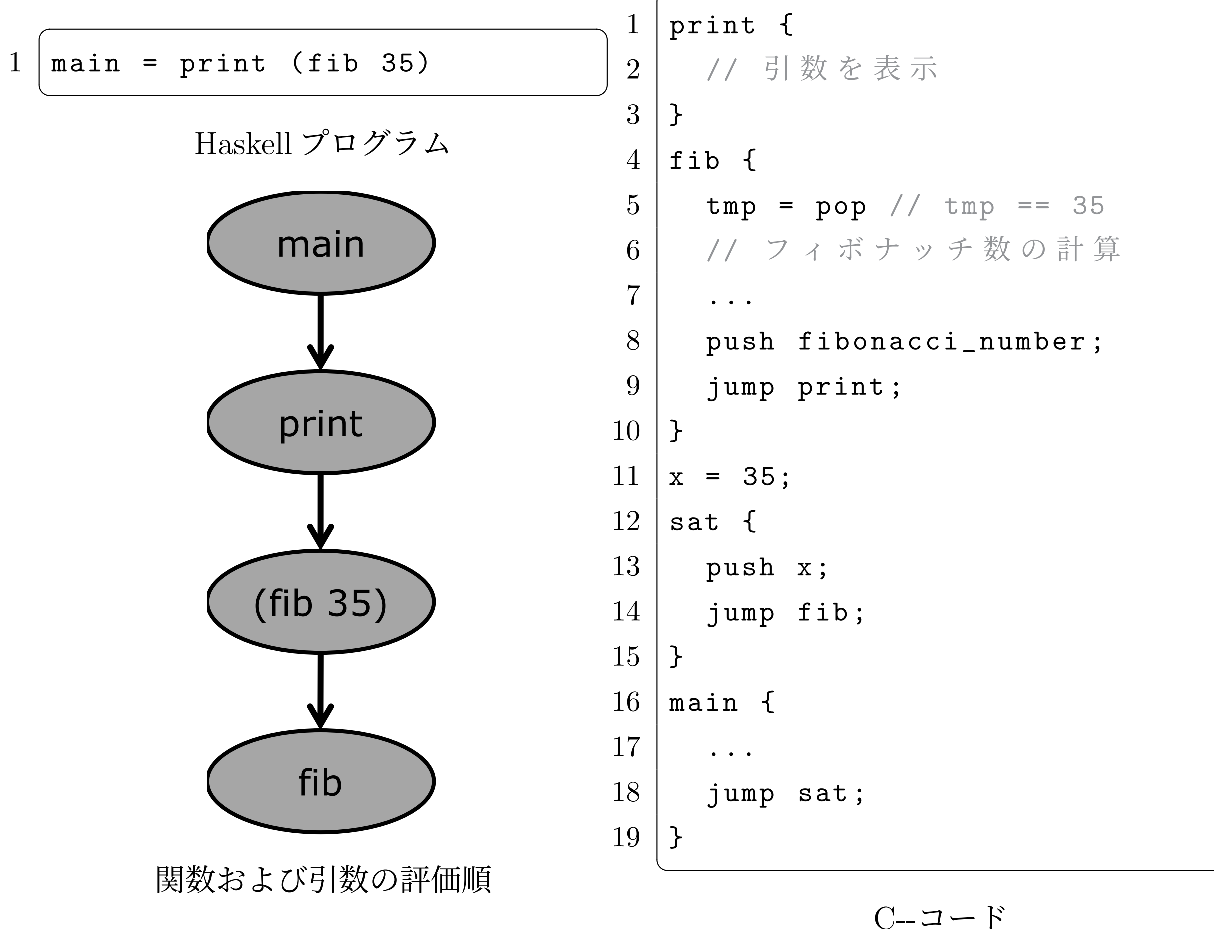


図 22: Haskell プログラムから C--コードへの変換例

このC--コードに関数をメモ化するためのコードを挿入する場合を考える．以降，本提案によって新たに挿入されるコードをメモ化コードと呼ぶ． C--コードに新たにメモ化コードを挿入したとき，その挿入されたメモ化コードはC--と同様にRTSによって実行される．そのため，関数の入出力を記憶するための領域の確保や，その入出力を記憶した入出力表を検索するための操作を図22で示したC--関数と同様に定義する必要がある． C--コードにメモ化コードを挿入する場合，任意の個所に挿入することができるため，任意の関数にのみメモ化を適用することができる．たとえば，図22の関数 `fib` にのみメモ化を適用したい場合には，C--コード中で関数 `fib` が呼び出されている個所にのみメモ化コードを挿入することで実現することができる．

次に，RTSを拡張しメモ化を実現する場合を考える．C--コードはRTSのSchedulerが発行したスレッドによって解釈・実行されるため，スレッドがC--コードの関数を実行する個所に拡張を行うことでメモ化を実現することができる． RTS に拡張を行う

```
1 int function (int x){
2   int y = foo(x);
3   return tailfunc(y);
4 }
```

図 23: 末尾呼び出し

ことでメモ化を実現した場合， C--コードのすべての関数をメモ化対象にすることができるが，メモ化による高速化が望めないような関数にもメモ化が適用されてしまう．しかし， C--コードに挿入したメモ化コードが RTS 上で実行されるのに対し， RTS を拡張しメモ化を実現する場合，メモ化を行うための操作は RTS と同様に実マシン上で実行されるため，より小さいオーバヘッドでメモ化を実現することができる．

5.3 引数の取得

C--コードの関数にメモ化を適用するには， C--コードが実行されるとき，その関数への入力がどこに存在するのかを正しく把握する必要がある．この際，GHC が Haskell プログラムを C--コードへと変換するとき， C--コードが末尾呼び出しと Push/Enter という二つの規則に従うように変換を行うという性質が利用できる．

末尾呼び出しとは，**図 23** に示す，関数 tailfunc の呼び出しにあたり，ある関数 function の戻り値を返す関数呼び出しである．このとき関数 `function` が実行されると，関数 `foo` が呼び出され，そして最後にその `y` を引数に関数 `tailfunc` が呼び出される．このように関数 `tailfunc` は `function` の最後に呼び出されるため，末尾呼び出しと呼ばれる． GHC では，Haskell プログラムから C--コードに変換される際に，すべての関数はこの末尾呼び出しになるように変換される．つまり， 図 22 の C--コードに示すように，関数呼び出しである `jump` が呼び出し元関数の末尾になるように変換される．これにより，GHC により変換された C--コードでは，関数の末尾以外に関数呼び出しがないことが保証される．

さらに，GHC が Haskell コードを C--コードへ変換するとき，Push/Enter という規則に従うように変換する． Push/Enter 規則に従った C--コードでは，関数呼び出しまでにその関数の実行に必要な引数がスタックにすべてプッシュされ，その後に関数の呼び出しが行われる．たとえば，図 22 に示すような Haskell プログラムの関数 `fib` を実行するとき，C--コードでは，`fib` の呼び出し元関数である `sat` 内で `fib` の実行に必要な引数 35 がスタックにプッシュされた後に，`fib` の呼び出しが行われていることが

わかる．このときスタックのトップから`fib`の引数が格納される． C--コードはこのような Push/Enter 規則に従ったコードであるため関数実行に必要な引数がすべてスタック上に存在している．

しかし，計算が遅延評価される Haskell では，C--コードの関数も遅延評価されるため，その関数の実行に必要な引数が未評価である可能性がある．そのため，単純にスタック上に存在するデータを関数の引数として取得することはできない．しかし，RTS が関数を実行する際の特徴を利用することでこの問題を解決することができる．

Haskell コードから変換された C--コードの関数は下記の 4 種類に分類することができ，その種類によって RTS が関数を実行する方法が異なる．

- Unknown function
- Known function, saturated call
- Known function, too few arguments
- Known function, too many arguments

GHC は Haskell プログラムをコンパイルする際に，関数が取る引数の数を解析しており，静的に解析可能な関数を Known function，静的に解析不可能な関数を Unknown function と分類している．さらに，Known function はその引数の状態によって 3 つに分類される．Known function, saturated call はコンパイル時にすでにその関数の実行に必要な引数の値が存在しているような関数である．たとえば，図 22 の関数`fib`に示すように，プログラム中に記述された即値を引数に取るような関数である．Known function, too few arguments と Know function, too many arguments は共に，コンパイル時には関数の実行に必要な引数が未評価で，実際の値がまだ存在しない関数であり，その引数の数によって分類されている．引数が六つ以下の関数は Known function, too few arguments に分類され，七つ以上の関数は Known function, too many arguments に分類される．

Known function, saturated call に該当する関数の実行に必要な引数はすべて評価された値であるため，RTS はこの関数を実行するとき，単純にスタックから値をポップして関数を実行する．よって Known function, saturated call にあたる関数をメモ化する場合，その関数が呼び出されている関数内でスタックにプッシュされた値を入出力表に記憶することで，メモ化対象関数の入力を記憶することができる．

一方，そのほかの 3 種類の Unknown function や Known function, too few arguments や Known function, too many arguments に該当する関数は実行に必要な引数の値が評価されていないため，関数を直ちに実行することはできない．そのため，RTS はヒー

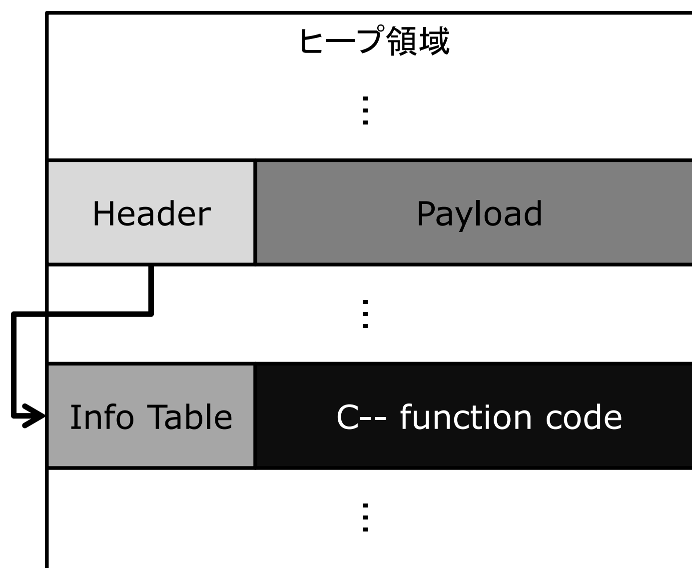


図 24: ヒープオブジェクト

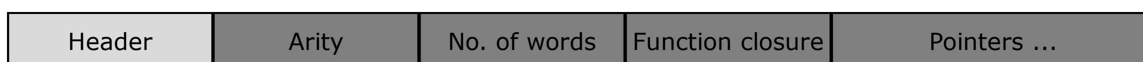


図 25: 関数オブジェクト

プ領域にその関数の情報を保持するための関数オブジェクトを作成し，関数の実行に必要な引数が評価されるたびにその関数オブジェクトを更新していき，すべての引数が評価されたときに関数を実行する．

この関数オブジェクトは RTS がヒープ領域上に作成するヒープオブジェクトの一つである．ヒープオブジェクトは**図 24** に示すような構造になっており， Header 部と Payload 部から構成されている． Header 部は C--コード中の関数および変数の情報を保持する Info Table へのポインタであり，Payload 部には関数の引数などオブジェクトの種類により様々なデータが格納される．

また，関数オブジェクトは**図 25** に示す構造になっており，各部には下記に示すデータが格納されている．

Header　Info Table へのポインタ

Arity　関数の実行に必要な引数の数

No. of words　Pointers 部のバイト数

Function closure　引数の一部を適用した関数のオブジェクトである Function closure オブジェクトへのポインタ

図 26: Function closure オブジェクト

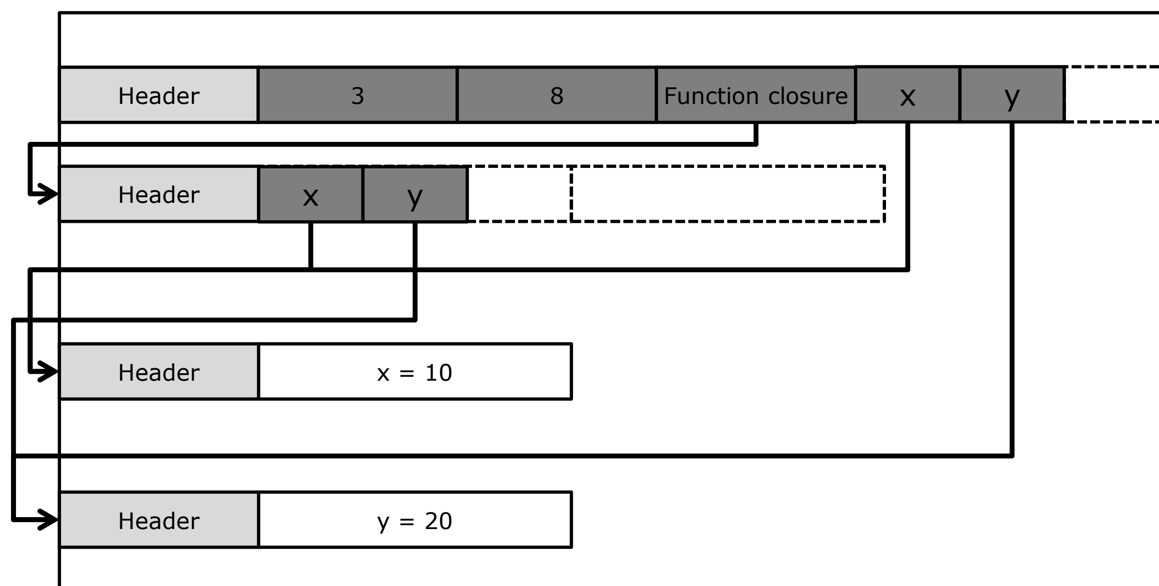


図 27: オブジェクトへの引数の適用

Pointers　すでに Function closure オブジェクトに適用された引数へのポインタ配列

また，Function closure オブジェクトは**図 26** に示す通りで，関数の実行に必要な引数へのポインタの配列またはその引数値の配列を Pointer 部または Non-pointer 部に保持している．関数の一部の引数をこの Function closure オブジェクトが保持することで未評価の関数を表現している． RTS は Known function, saturated call 以外に該当する関数ごとに図 25 に示す関数オブジェクトを作成する．そして，その関数の引数が評価されるたび，その関数オブジェクトの Pointer 部にその引数へのポインタを追加し，同時に Function closure オブジェクトの Pointer 部または Non-pointer 部に引数へのポインタまたは引数の値を追加する．

例えば，三つの引数を取る関数の関数オブジェクトには**図 27** に示すように，Arity 部には引数の数 3 が格納されている．この例は，三つの引数の内，二つが評価されたときの，関数オブジェクトと Function closure オブジェクトの状態を示している．関数オブジェクトの Pointer 部には二つのポインタが格納されており，それぞれ第一引数（x = 10），第二引数（y = 20）を指している．また，No. of words 部には Pointer 部のバイト数である 8 が格納される．実行環境により RTS で表現されるポインタのバイ

ト数が異なるが，この例ではポインタは 4 バイトで表現されている．Function closure オブジェクトも同様に Pointer 部に引数へのポインタを格納する．ここで三つ目の引数が評価されると，関数オブジェクトの Pointer 部に新たに第三引数へのポインタが追加され，No. of words 部の値が 12 に更新される．さらに，Function closure オブジェクトの Pointer 部にも引数へのポインタが追加される．

Known function, too few arguments に該当する関数の関数オブジェクトの Arity 部には，関数の実行に必要な引数の数が格納されており，Pointer 部には評価された引数へのポインタが格納されている．そのため，Arity 部の値と Pointer 部に格納されるポインタの数が一致したとき，関数の実行に必要なすべての引数の値が評価されたことを検出することができる．このため，Known function, too few arguments に該当する関数にメモ化を適用する場合，Arity 部の値と Pointer 部に格納されるポインタの数が一致したときに，図 26 に示した Function closure オブジェクトの Pointer 部および Non-pointer 部に格納される値を入出力表に記憶することで，関数の入力を入出力表に登録することができる．

一方，Unknown function および Known function, too many arguments に該当する関数はコンパイル時の静的解析により，実行に必要な引数の数を取得することができず，関数オブジェクトの Arity 部に格納される値は常に 0 である．これらの関数にメモ化を適用しようとしたとき，いつ全ての引数の値が評価されたかを検出することができないため，メモ化対象から外すこととする．

5.4 入出力表

5.4.1 入出力表領域の確保

関数にメモ化を適用する場合，その関数の入力および出力を記憶するための入出力表が必要になる．この入出力表のための領域を Storage Manager によって管理されるヒープ領域に確保する場合と，実マシン上のメモリ領域に確保する場合が考えられる．

入出力表領域を RTS の Storage Manager によって管理されるヒープ領域に確保する場合，入出力表もヒープオブジェクトと同様に Storage Manager によって管理されることになる．そのため，入出力表にアクセスするとき Storage Manager を介してアクセスすることになり，オーバヘッドが発生する．また，Storage Manager はガベージコレクションによってヒープ領域上の不要となったオブジェクトが使用していた領域を自動的に解放する．そのため，メモ化対象関数が呼び出される可能性のある間は，その関数の入出力表によって使用される領域がガベージコレクションによって解放され

```
1 sat   : [x, fib, print]
2 fib   : [print]
3 print : []
4 main  : [print, fib, sat]
```

```
1 sat   : [x, fib, print]
2 fib   : [print, memotable]
3 print : []
4 main  : [print, fib, sat]
```

図 28: Static Reference Table

ないようにする必要がある．

C--コード中の関数はその関数が呼び出された時に実行されるコードとは別にその関数の引数や戻り値の型など関数の実行に必要な情報を保持する Info Table を持っている．Info Table には，関数が参照する変数や呼び出される関数の配列である Static Reference Table（SRT）が格納されている．図 22 の右に示すような C--コードの場合，各関数の SRT は**図 28** 左に示すよな構造になっており，1 行目の SRT は関数 `sat` が `x`，`fib`，`print` を参照することを示している．このとき変数 `x` は SRT によって，`sat` からしか参照されていないことがわかる．関数 `sat` の実行が終了したとき，`x` はどこからも参照されなくなり，RTS の Storage Manager によって `x` が使用していた領域は解放される．

たとえば，図 22 の `fib` にメモ化を適用したい場合，その入出力を記憶した入出力表の領域をヒープ領域上に確保する．このとき，図 28 右に示すように関数 `fib` の SRT に入出力表 `memotable` を追加することで，`fib` が呼び出される可能性のある間は，入出力表領域が解放されるのを防ぐことができる．また，`fib` が呼び出されることが無くなると，ガベージコレクションによって自動的に入出力表領域が解放されるため，ヒープ領域に不要となった入出力表領域が存在し続けることを防ぐことができる．

次に実マシンのメモリ上に入出力表領域を確保する場合を考える．実マシンのメモリ領域は Storage Manager によって管理される領域では無いため，RTS から直接アクセスすることができる．そのため，Storage Manager を介してアクセスするヒープ領域に比べアクセスのオーバヘッドが小さく入出力表の検索および値の登録をより高速に行うことができる．しかし，RTS 上で実行される C--コードから実マシンのメモリに直接アクセスすることはできないため，メモ化コードを C--コードに挿入する手法でメモ化を実現することはできない．

さらに，実マシンの上のメモリに入出力表領域を確保したとき，その領域は RTS の Storage Manager に管理されないため，必要で無くなった場合でもガベージコレクションによって自動的に解放されない．そのため，入出力表領域がメモリ上に存在し続け，

```
1  fib {
2      tmp = pop // tmp == 35
3      // フィボナッチ数の計算
4      ...
5      push fibonacci_number
6      regOutVal fibonacci_number // 出力値登録コードの挿入
7      jump print;
8  }
9  x = 35;
10 sat {
11     push x;
12     regInVal x // 入力値登録コードの挿入
13     jump fib;
14 }
```

図 29: Known function, saturated call の入出力登録

メモリを圧迫する可能性がある．

5.4.2　入出力表への値の登録

関数にメモ化を適用したいとき，その関数の入力および出力を入出力表に登録する必要がある．メモ化対象関数の入力である引数は 5.3 節で述べた方法によって取得することができる．また，C--コードでは関数間での値の受け渡しを行うのにスタックを用い，呼び出される関数に値を渡したい場合，その値をすべてスタックにプッシュする．そのため，関数の実行中にスタックにプッシュしたすべての値がその関数の出力値である．例えば，図 22 の関数 `fib` の場合，その出力は関数実行中にスタックにプッシュされた `fibonacci_number` の値である．

ここで，C--コードにメモ化コードを挿入することでメモ化を実現する場合を考える．メモ化対象関数が**図 29** の `fib` に示すような Known function, saturated call の場合，5.3 節で述べたように，関数が呼び出されるまでに，その関数の実行に必要な引数がすべて評価された値としてスタックにプッシュされている．そのため，図 29 の 12 行目に示すように，関数呼び出しの直前に，スタックへプッシュされた値を登録するコードを挿入することで関数の入力を入出力表へ登録することができる．そして，GHC によって C--コードの関数はすべて末尾呼び出しになっているため，関数実行中にスタックへプッシュされた値がその関数の出力値である．そのため，図 29 の 6 行目に示すよ

うに，関数`fib`中でスタックにプッシュされた値を登録するコードを挿入することで関数の出力を入出力表へ登録することができる．

一方，Known function, too few arguments にあたる関数にメモ化を適用したい場合， 5.3 節で述べたように関数の実行に必要な引数の値が評価されていない場合があり，関数オブジェクトを参照することで引数を得ることはできるが，関数オブジェクトは RTS の Storage Manager によって管理されるヒープ領域上に存在するため，C--コードからそのオブジェクトを参照することはできない．このため，C--コードにメモ化コードを挿入する場合， Known function, too few arguments の出力値を入出力表に登録することはできない．

次に，RTS を拡張しメモ化を実現する場合を考える．メモ化対象関数が Known function, saturated call の場合は，C--コードにメモ化コードを挿入する場合と同様に，メモ化対象関数の呼び出し直前にスタック上の値を入力値として登録し，メモ化対象関数末尾の関数呼び出し直前にスタック上の値を出力値として登録することで，入出力表に関数の入出力を登録することができる．一方，Known function, too few arguments に該当する関数にメモ化を適用する場合， RTS の Storage Manager によって管理されるヒープ領域上の関数オブジェクトを参照することで関数の実行に必要な引数を得ることができる．また，関数の引数がすべて評価されて関数が実行可能になると RTS は関数のコードを実行し，Known function, saturated call の場合と同様に出力値をスタックにプッシュする．このため，出力値の登録は Known function, saturated call と同様に，メモ化対象関数末尾の関数呼び出し直前にスタック上の値を出力値として登録することで，入出力表に関数の入出力を登録することができる．

5.4.3 入出力表の検索

関数にメモ化を適用する場合，メモ化対象関数が呼び出されるとき，その関数の入力で入出力表を検索する必要がある．この方法として，C--コードで検索を行う関数を記述しその実行を RTS 上で行う方法と， RTS を拡張し RTS 自身が入出力表の検索を行う方法が考えられる．

C--コードで入出力表の検索を行う関数を記述した場合，この関数はメモ化対象関数が呼び出される直前に実行されることになる．そのため，メモ化対象関数が Known function, saturated call の場合，検索関数が入出力表を検索するのに必要とするメモ化対象関数の引数はすでにスタック上にプッシュされており，新たに引数を取得し，スタックにプッシュする必要がない．しかし，Known function, too few arguments にあたる関数をメモ化する場合，その引数の値は評価されていない可能性があるため，検

表 2: 評価環境

CPU	Core 2 Extreme Q6850
動作周波数	3.00GHz
コア数	4
メモリ	8GB
OS	Solaris 10
コンパイラ	Glasgow Haskell Compiler
バージョン	6.12.2
最適化オプション	-O0

索を行うことができない場合がある．また，この検索関数は他の C--関数と同様に RTS 上で実行されるため，オーバヘッドが発生する．

一方，RTS を拡張し，RTS が直接入出力表の検索を行う場合，メモ化対象が Known function, too fewarguments の場合であっても，引数の値がすべて評価されているかどうか判断することができるため，Known function, too few arguments がメモ化対象の場合も入出力表の検索を行うことができる．また，C--コードで記述された検索関数は RTS 上で実行されるのに対し，RTS を拡張した場合の入出力表検索は実マシンによって直接実行されるため，入出力表検索のオーバヘッドが小さく，高速に行うことができる． p

6 評価

4 章で示した各手法を評価プログラムに適用し，評価プログラムの実行時間を計測し，その比較を行った．

6.1 評価環境

評価環境は**表 2** に示す通りで，コンパイラには GHC を用いた．コンパイラによって，本提案であるメモ化以外の最適化が働かないようにするために，最適化オプションは-O0 とした．

既存手法との比較を行うために再帰関数であるフィボナッチ数を求めるプログラムとたらい回し関数を評価プログラムとして用いた．また，再帰関数以外の評価プログラムとして，入力画像をグレースケール化するプログラムと高速フーリエ変換を行う

```
1 main = do let a = fib 45
2               b = fib 10
3               c = fib 17
4               d = fib 35
5               e = fib 5
```

図 30: フィボナッチ数を求めるプログラム

```
1 tak :: Int -> Int -> Int -> Int
2 tak x y z = if (x <= y)
3                 then z
4                 else tak xn yn zn
5                        where xn = tak x-1 y z
6                              yn = tak y-1 z x
7                              zn = tak z-1 x y
```

図 31: たらい回し関数

プログラムを用いた．フィボナッチ数を求めるプログラムでは**図 30** で示すように，関数 fib を異なる引数で 5 回呼び出し，すべての実行が終了するまでの時間を計測した．たらい回し関数は**図 31** に示す関数で，関数に与える引数によって再帰呼び出し回数が非常に多くなる．たらい回し関数 `tak` は Haskell のベンチマークプログラムである The nofib Benchmark Suite of Haskell Programs [10] のたらい回し関数と同じ (`33，17，8`) を引数に与えたときの実行時間を測定した．また，入力画像をグレースケール化するプログラムは一画素をグレースケール化する関数をメモ化対象とした．入力画像には 1920×1080 の大きさの画像を使用した．高速フーリエ変換を行うプログラムは $\theta = 2\pi/n * k(k : 0 \to (n-1))$ を求める関数が同一の n を引数にとり複数回呼び出されるため，この関数をメモ化対象とした．フィボナッチ数を求める関数とたらい回し関数は共に再帰関数であるため 3.1 節の既存手法を適用することができるが，入力画像をグレースケール化するプログラムは再帰関数でないため既存手法を適用することはできない．さらに，汎用 Genetic Algorithm (GA) ソフトウェアである GENEsYs の適応度関数にメモ化を適用し，その実行時間の計測を行った．

また， 4.3.1 項の外部関数呼び出しによる入出力表検索の高速化と，4.3.2 項の検索と関数実行の並列化による検索オーバヘッドの隠蔽は，IO モナドを用いた関数をメモ

表 3: 再帰関数

	フィボナッチ関数	たらい回し関数
通常実行	115.818164s	9.667706s
既存手法	0.000782s	0.003962s
提案手法 1	0.000442s	0.003030s
提案手法 1 (外部関数)	0.000384s	0.001768s
提案手法 1 (並行化)	0.003797s	0.062288s

表 4: 非再帰関数

	グレースケール化	高速フーリエ変換
通常実行	0.063169s	1.046818s
既存手法	N/A	N/A
提案手法 1	3.185307s	0.579159s
提案手法 1 (外部関数)	2.273859s	0.578166s
提案手法 1 (並行化)	N/A	0.667885s

化する際にしか適用することができない．そのため，これらの手法を評価するために，評価プログラムのメモ化対象関数を IO モナドを用いた関数へ書き換えを行った．

6.2 実行時間の比較

通常実行，既存手法を適用した場合，本提案手法である自動変換によるメモ化を適用した手法，外部関数呼び出しによる入出力表検索の高速化を適用した手法，入出力表検索と関数実行の並行化を適用した手法で，評価プログラムの実行時間を測定し，比較を行った．評価プログラムに各手法を適用したときの実行時間を**表 3** および**表 4** に示す．表中では各手法をそれぞれ既存手法，提案手法 1，提案手法 1（外部関数），提案手法 1（並行化）と表している．

フィボナッチ数を求めるプログラムでは，メモ化を適用することにより，通常実行した場合に比べ，高速化することができた．さらに，図 30 で示すように，フィボナッチ数を求めるプログラムは，異なる引数で関数 `fib` を複数回呼び出している．既存手法では，再帰関数内でしか関数 `fib` の入出力を記憶した入出力表を共有していないため，過去の `fib` 呼び出しで実行した結果を再利用できない．これに対し，提案手法 1

表 5: 既存手法および提案手法 1 適用時のたらい回し関数実行時間の内訳

既存手法	割合	実行時間	提案手法 1	割合	実行時間
gmTak	4.9%	0.000194s	tak	3.4%	0.000148s
memo	8.6%	0.000341s	tak_wrap	1.9%	0.000058s
mapDict	9.5%	0.000376s			
lookup	45.0%	0.001783s	lookup	57.7%	0.001748s
insert	32.0%	0.001268s	insert	37.0%	0.001121s

では，ラッパー関数の引数と戻り値を介して複数の `fib` 呼び出し間で入出力表を共有しているため，過去の実行結果を再利用することができ，既存手法に対し，約 1.77 倍の高速化を実現した．さらに，提案手法 1（外部関数）では，C 言語で記述した入出力表操作関数を用たことで，既存手法に比べ約 2.03 倍，提案手法に比べ約 1.15 倍の高速化を実現することができた．

また，たらい回し関数の場合も，通常実行時に比べメモ化を適用することにより高速化することができた．さらに，既存手法に比べ，提案手法 1 を適用した場合，約 1.31 倍の高速化を実現している．これは，既存手法では再帰関数内の再帰関数呼び出しの間で入出力表を共有するために State モナドを用いたことによってオーバヘッドが発生し既存手法の実行時間が増加したためと考えられる．そこで，RTS の Profiler によって，既存手法を適用したプログラムと提案手法 1 を適用したプログラム中の関数の実行時間を調査した．プログラム全体の実行時間に対する各関数の実行時間の割合と実行時間を**表 5** に示す．表 5 に示す通り，既存手法，提案手法 1 ともに入出力表を検索する `lookup` 関数と入出力表に値を登録する `insert` 関数が実行時間の多くを占めている．既存手法では入出力表の受け渡しを隠蔽するために State モナドを用いていた．既存手法の `mapDict` に要した時間は入出力表の検索と値の登録関数全体の実行時間から，実際に検索や値の登録を行う `lookup` 関数と `insert` 関数の実行時間を引いたものである．つまり，State モナドによるオーバヘッドを表している．提案手法 1 では State モナドを用いていないため，このオーバヘッドは発生していない．提案手法 1 と比較すると既存手法は State モナドのオーバヘッド `mapDict` と `memo` によって実行時間が増加している．既存手法では `fix` によって関数中の再帰呼び出しをメモ化適用済の関数に置き換えていた．そのため，`fix` による関数の置き換えオーバヘッドが発生し，`memo` の実行に時間を要している．提案手法 1 ではこれらのオーバヘッドが発生しないため，

表 6: 提案手法 1 適用グレースケール化プログラムの実行時間の内訳

関数名	呼び出し回数	割合	実行時間
gray_wrap	2073600	99.64%	3.397741s
gray	83491	0.36%	0.012276s

表 7: 理想状態での実行時間の比較

	実行時間
通常実行	0.062816s
提案手法 1	0.563852s

既存手法に比べ高速化を実現することができていることが分かる．

一方，グレースケール化プログラムでは，メモ化を適用した場合，通常実行時よりも動作速度が低下してしまった．たらい回し関数と同様に RTS の Profiler により提案手法 1 を適用したときの各関数の実行時間を計測した，計測結果を**表 6** に示す．`gray_wrap` はラッパー関数，`gray` は 1 画素をグレースケール化する関数を示している．メモ化を適用しない場合，`gray` の呼び出し回数は 1920×1080 サイズの画像のため 2073600 回になるが，メモ化を適用したことによって 83491 回に減っている．`gray` の実行時間だけを見ると表 4 の通常実行の実行時間に比べ短縮されていることがわかる．しかし，ラッパー関数の実行時間が，短縮された時間に対し大きくなってしまったことによって，全体の実行時間が増加してしまっている．

ここで，入力画像に単色画像（全ての画素が同じ画素値を持つ画像）を使用し，既存手法と提案手法 1 の実行時間の比較を行った．この場合，メモ化対象関数が最初の 1 度しか呼び出されず，のこりの呼び出しはすべて再利用される．さらに，入出力表には 1 つの入出力しか登録されないため，検索のオーバヘッドも最も少なくなる．測定結果を**表 7** に示す．理想的な入力を用いたことで，提案手法 1 を適用した場合，検索時間が短縮され表 4 に示した場合に比べ約 5.65 倍の高速化を実現しているが，依然として通常実行時よりも実行時間が増加している．これは，メモ化を適用することによって発生する入出力表検索オーバヘッドが，計算結果を再利用することで短縮される時間に比べ大きくなってしまったためであると考えられる．このため，メモ化対象関数の実行時間が比較的小さい場合には，メモ化による高速化が望めない．

また，高速フーリエ変換プログラムに提案手法 1 を適用した場合，表 4 に示すよう

表 8: 並行化手法適用時の実行時間割合とメモリ利用割合

関数名	実行時間割合	メモリ利用割合
forkIO	17.93%	87.62%
Found	15.30%	2.00%
NotFound	51.59%	8.00%
newEmptyMVar	1.25%	1.40%
takeMVar	13.93%	0.78%

に，通常実行時に比べ約 1.80 倍の高速化を実現した．一方，提案手法 1（外部関数）を適用した場合，提案手法 1 に比べ高速化を実現できているが，フィボナッチ関数やたらい回し関数に比べるとその高速化率は小さい．これは，今回メモ化対象関数とした関数が同一の n を引数に複数回呼び出されており，入出力表に登録される入出力の数が一つであったため，提案手法 1 を適用した場合でも入出力表の検索オーバヘッドが十分小さく，高速であったためである．

しかし，提案手法 1（並行化）では，提案手法 1 と比べ実行時間が増加してしまった．さらに，グレースケール化プログラムでは提案手法 1（並行化）を適用したときプログラムの実行時間が増加しすぎてしまい，現実的な時間では終了しなかった．そこで，フィボナッチ数を求めるプログラムを 1 スレッドで実行し，そのときのスレッドの作成と，同期変数の作成および同期変数から値を取得するのに要した時間の割合とその操作に利用されたメモリの割合を測定した．その測定結果を**表 8** に示す．forkIO はスレッドの作成，Found は検索スレッドで実行される処理，NotFound は関数実行スレッドで実行される処理，newEmptyMVar は同期変数の作成，takeMVar は同期変数からの値の取得を表しており，それぞれの実行時間の割合と利用されたメモリ領域の割合を示している．

スレッドの作成時間 forkIO と同期変数の作成 newEmptyMVar および値の取得 takeMVar に要する時間が，検索と関数実行の並行化によって隠蔽される，検索スレッドの実行時間 Found に比べ大きな割合となってしまった．そのため，並行化を適用したことにより適用しなかった場合にくらべ実行時間が増加してしまった．また，メモリ利用割合を見ると，利用されるメモリ領域のほとんどがスレッドの作成 forkIO によって利用されている．GHC によってコンパイルされたプログラムは一定の期間ごとにガベージコレクションによって必要とされないデータのメモリ領域が解放される．並行

表 9: 適応度関数の実行時間 (Double)

	f_01	f_02	f_03	f_06
通常実行	26.715175s	53.154232s	70.751601s	31.458703s
提案手法 1	26.421130s	22.448188s	22.075597s	26.561166s

f_07	f_15	f_21	f_22	f_23
48.466535s	35.680029s	49.032402s	49.051426s	29.596733s
25.259524s	29.071589s	26.484271s	26.491100s	20.425296s

表 10: 適応度関数の実行時間 (Bool)

	f_12	f_14
通常実行	11.424686s	12.296546s
提案手法 1	17.150884s	17.169985s

化手法を適用した場合，メモ化対象関数が呼び出されるごとに 2 つのスレッドが作成され，実行終了時に両スレッドともに破棄される．そして，作成したスレッドによって利用されたメモリ領域が解放されるのはガベージコレクションが行われるタイミングである．2073600 回のメモ化対象関数呼び出しを含むグレースケールプログラムでは 2073600 $\times$ 2 つのスレッドが作成されることになる．そのため大量に作成されたスレッドが利用していたメモリ領域がガベージコレクションのタイミングで一度に解放されるため，ガベージコレクションに要する時間が大きくなり，現実的な実行時間で終了しなかったと考えられる．

次に，汎用 GA ソフトウェアである GENEsYs にメモ化を適用しその実行時間を計測した． GA ではデータを遺伝子として表現した個体を複数用意し，それらの個体を交叉させ，新たに得られた個体に対して，その個体の適応度を定量化する適応度関数を用いて他の個体に比べどれだけ優れているのかを計算する．これらの操作を繰り返し行い，最も適応度の高い個体を解として出力する．今回の実験では，個体を 50 用意し，各個体の長さは 32 とした．交叉と適応度関数の適用の繰り返しを合計 1000 回行い，その実行時間の測定を行った．これらの値は GENEsYs の初期値を用いた．また `Double` 型の値のリストを引数に取る適応度関数 `f_01`，`f_02`，`f_03`，`f_06`，`f_07`，`f_15`，`f_21`，`f_22`，`f_23` と `Bool` 型の値のリストを引数に取る適応度関数 `f_12`，`f_14` にメモ化を適用した．メモ化対象とした適応度関数はリストを引数に取るため，FFI を用い

表 11: f_03 の実行時間の内訳

	関数名	呼び出し回数	割合	実行時間
通常実行	f_03	2447600	91.2%	64.524548s
提案手法 1	f_03_wrap	2447600	68.7%	15.165935s
	f_03	5286	0.9%	0.198680s

表 12: f_12 の実行時間の内訳

	関数名	呼び出し回数	割合	実行時間
通常実行	f_12	2447600	34.7%	3.964366s
提案手法 1	f_12_wrap	2447600	67.6%	11.593997s
	f_12	3447	0.1%	0.017150s

て C 言語記述した関数に引数のリストを渡すことができない．そのため，入出力表操作のオーバヘッド削減手法を適用することはできない．それぞれ適応度関数を用いたときの実行時間を**表 9** と**表 10** に示す．

表 9 に示すように `Double` 型の値のリストを引数に取る適応度関数にメモ化を適用した場合では，提案手法 1 により自動的にメモ化を適用したことにより高速化を実現することができた．適応度関数 `f_03` にメモ化を適用した場合では約 3.20 の高速化を実現することができた．適応度関数 `f_03` にメモ化を適用したときの実行時間の内訳を**表 11** に示す．通常実行では全体の実行時間の 91.2%を適応度関数 `f_03` が占めていた．一方，提案手法 1 を適用したことにより，`f_03` の呼び出し回数が 2447600 回から 5286 回に減少している．また，ラッパー関数によって増加した時間に比べ，`f_03` にメモ化を適用したことによって減少した時間が大きかったため，高速化を実現することができた．

一方で，表 10 に示すように `Bool` 型の値のリストを引数に取る適応度関数にメモ化を適用したときは実行時間が増加してしまった．適応度関数 `f_12` にメモ化を適用した時の実行時間の内訳を**表 12** に示す．提案手法 1 を適用したことにより `f_12` の呼び出し回数は 2447600 回から 3447 回に減少し，実行時間も同様に小さくなっている．しかし，減少した実行時間に比べ，メモ化を適用したことによって発生するラッパー関数の実行時間が大きくなってしまったため，低速化してしまったと考えられる．これは，今回メモ化対象とした適応度関数 `f_12` の処理量が小さかったため，メモ化によって省

略できる実行時間が小さくなってしまったためである．

7　おわりに

7.1　まとめ

本論文では，関数型言語 Haskell で記述されたプログラムに自動的にメモ化を適用する二つの手法を提案した．その一つ目として，Haskell プログラムの自動変換によるメモ化手法をを提案した．本手法では，プログラマによって指定されたメモ化対象関数ごとに入出力表の検索や入出力表へ値の登録を行うラッパー関数を生成し，対象関数の呼び出しをラッパー関数の呼び出しへ自動変換することでメモ化を実現した．さらに，対象関数を呼び出していた関数の引数および戻り値に，新たに入出力表を追加し，対象関数とその呼び出し元関数の間で入出力表を受け渡し可能にした．このように変換することで，複数の関数呼び出しの間で再利用表を共有し，既存手法では対応していなかった非再帰関数に対してもメモ化を適用可能にした．

また，メモ化を適用した際に問題となる，入出力表操作のオーバヘッドを削減する二つの手法をプログラム変換時に自動的に適用することでラッパー関数の実行時間の短縮を図った．一つめの高速化手法として，関数型言語 Haskell に比べより実行速度の高速な C 言語で記述された入出力表の検索関数および値の登録関数を自動生成し，これらの C 言語で記述された関数をを Haskell プログラムから呼び出すことでラッパー関数の高速化を行う手法を提案した．さらに，ラッパー関数内で二つのスレッドを作成し，入出力表の検索と関数実行を並行に行うことで，検索が失敗した時の検索オーバヘッドを隠蔽する手法を提案した．

さらに，これらの手法の有効性を探るため，Haskell プログラムに各手法を適用し，その実行時間を測定し，比較を行った．その結果，既存手法を適用した場合に比べ，Haskell プログラムの自動変換によるメモ化手法は最大で約 1.77 倍の高速化を実現した．さらに，外部関数呼び出しによる高速化手法を適用した場合，既存手法に対し，最大で約 2.03 倍の高速化を実現した．また，汎用 GA ソフトウェアである GENEsYs に提案手法を適用した場合では，最大で約 3.20 倍の高速化を実現し，その有効性を示した．

さらに，Haskell プログラムに自動的にメモ化を適用する二つ目の手法として，Haskell コンパイラである GHC を拡張し，プログラムのコンパイル過程で中間コードおよびランタイムシステムにメモ化を実現するコードを自動挿入する手法を検討した．対象関数にメモ化を適用する際に必要となる関数の入力を，中間コードの末尾呼び出しや

Push/Enter などの規則性およびランタイムシステムの関数実行方法の特徴を用いることで取得可能であることを示した．

また，入出力表領域を確保するための領域としてランタイムシステムによって管理されるヒープ領域と実マシン上のメモリ領域が考えられるが，これらの領域に確保したときの，入出力表の検索方法や値の登録方法について検討し，ヒープ領域上に入出力表領域を確保した場合には，Haskell のランタイムシステムによってその入出力表領域も管理させることで，無駄な領域をガベージコレクションによって自動的に解放させることができることを示した．

7.2 今後の課題

本研究の今後の課題として，検索と関数実行の並行化手法の高速化があげられる．6.2 節で述べたように，この手法を適用したとき，ラッパー関数の呼び出しが行われるたび，検索スレッドと関数実行スレッドの二つのスレッドと同期変数が作成される．そのため，スレッドの作成および同期変数の作成に要する時間により高速化が実現できていなかった．それゆえ，これらの時間を短縮することができれば高速化を実現できると考えられる．そこで，その方法として検索スレッドと関数実行スレッドの二つのスレッドを作成するのではなく，一方のスレッドにメインスレッドを利用する方法が考えられる．この場合，スレッドの作成時間が半分になり，高速化が実現できると考えられる．さらに，作成されるスレッド数が抑制され，ガベージコレクションによって一度に解放されるメモリ領域も減少し，より多くのプログラムに対して並行化手法が適用可能になると考えられる．さらに，ラッパー関数が呼び出される度にスレッドの作成を行うのではなく，一度作成したスレッドを再び利用できるようにすることで，スレッドの作成時間をより短縮できると考えられる．

また，外部関数の呼び出しによる高速化手法を適用する際，関数の引数が `Int` 型または `String` 型以外の型の場合，本研究では入手力表の検索を行うためのキー値を求める関数をプログラマに記述して貰うこととした．これは，引数の取る値の範囲の特定がこんなんであったためである．そこで，プログラマにプラグマなどを用いて，引数の取る値の範囲を記述して貰うことでキー値を求める関数を自動生成することが可能になり，この関数の記述を省くことができると考えられる．

さらに，IO モナドを用いた関数にメモ化を適用したい場合には，プログラマにプラグマを用いて，その関数が参照透過性を備えていることを保証して貰う必要があった．これは，その関数中で入出力処理を行っているかどうか判定することが困難であった

ためである．しかし，Haskell のライブラリで入出力を行う関数は限られているため，これらの入手力関数がメモ化対象関数から呼び出されているかどうか調査することで，参照透過性を備えているかどうか判定することが可能である．

また，評価では処理量の小さい関数にメモ化を適用してしまい，省略できる実行時間に比べ，メモ化のオーバヘッドが大きかったため，全体の実行時間が増加してしまっていた．そこで，処理量が小さい関数などメモ化による高速化が望めない関数には，自動的にメモ化を適用しないようにすることが考えられる．

また，本研究で提案したコンパイラ拡張によるメモ化手法はその実現方法を検討するに止まっている．そこで今後，この手法を実際に GHC に実装し，入出力表領域を確保する領域や入出力表の検索方法の違いを検討したいと考えている．

謝辞

本研究のために多大な御尽力を頂き， 日頃から熱心な御指導を賜った名古屋工業大学の松尾啓志教授，津邑公暁准教授， 齋藤彰一准教授, 松井俊浩助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室ならびに齋藤研究室の皆様に深く感謝致します.

参考文献

[1] Hudak, P. and Jones, M. P.: Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity, Technical report (1994).

[2] Hartel, P., Feeley, M., Alt, M., Augustsson, L., Baumann, P., Beemster, M., Chailloux, E., Flood, C., Grieskamp, W., Van Groningen, J. et al.: Benchmarking implementations of functional languages with ’Pseudoknot’ , a float-intensive benchmark, *Journal of functional programming*, Vol. 6, No. 04, pp. 621–655 (1996).

[3] Jones, S. P.: *Haskell 98 Language and Libraries*.

[4] Marlow, S.: *Haskell 2010 Language Report*.

[5] O’Sullivan, B., Goerzen, J. and Stewart, D.: *Real World Haskell*, O’Reilly (2009).

[6] Brown, D. and Cook, W. R.: Function Inheritance: Monadic Memoization Mixins, *Proc. Brazilian Symposium on Programming Lnaguage* (2009).

[7] Chakravarty, M., Finne, S., Henderson, F., Kowalczyk, M., Leijen, D., Marlow, S., Meijer, E., Panne, S., Jones, S. P., Reid, A., Wallace, M. and Weber, M.: *The Haskell 98 Foreign Function Interface 1.0*.

[8] University of Glasgow: *Glasgow Haskell Compiler*, http://haskell.org/ghc/.

[9] Ramsey, N., Jones, S. P. and Lindig, C.: *The C-- Language Specification Version 2.0* (2005).

[10] Partain, W.: The nofib Benchmark Suite of Haskell Programs, *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, London, UK, Springer-Verlag, pp. 195–202 (1993).