情報処理学会論文誌　プログラミング　Vol.5　No.2　67–78 (Mar. 2012)

Glasgow Haskell Compilerにおける再帰的データ構造のための遅延オブジェクトの再利用

高野 保真[1,a)]　岩崎 英哉[2]　鵜川 始陽[2]

受付日 2011年10月1日, 採録日 2011年11月23日

概要：遅延評価は，プログラムの簡潔な記述を可能にするため，純関数型言語などで採用されている．特にリストなどの再帰的データ構造を扱う際には，必要に応じて評価を進めることが可能となり，遅延評価により得られる記述性の向上は大きい．一方，計算を遅延するために必要なオブジェクト（遅延オブジェクト）の生成は，プログラム実行時のオーバヘッドとなってしまうため，効率の良い遅延評価機構を実装するには，遅延オブジェクトの削減が必要である．本論文は，リストのような線形に再帰的に定義される代数データ構造に注目し，必要となる遅延オブジェクトを再利用する手法を提案する．提案手法は，すでに割り当てられている遅延オブジェクトの保持している値を更新して再利用する．このような再利用を可能とするために，提案手法は，コンパイル時にプログラム変換を行い，再利用対象とする遅延オブジェクトへの参照を単一にする．提案手法を，純関数型言語 Haskell の処理系である Glasgow Haskell Compiler 上に実装し，実験を行った．その結果，オーバヘッドはあるものの，総メモリ割当て量を削減できることが分かった．

キーワード：遅延評価，Haskell，関数型言語処理系，再帰的データ構造

Reusing Thunks for Recursive Data Structures in Glasgow Haskell Compiler

Yasunao Takano[1,a)]　Hideya Iwasaki[2]　Tomoharu Ugawa[2]

Received: October 1, 2011, Accepted: November 23, 2011

Abstract: Lazy evaluation helps programmers write clear programs. However, it has significant run-time overheads for building many as-yet unevaluated expressions, or thunks. Because thunk allocation is a space-consuming task, it is important to reduce the number of thunks in order to improve the performance of a lazy functional program. We propose static analysis algorithms that achieve the thunk reuse technique. Thunk generation is suppressed by reusing an already allocated thunk at the tail of a list, on the condition that the thunk is singly referred, i.e., pointed to only from the tail field of a cons cell. This method guarantees that reused thunks definitely satisfy this singly referred condition on the basis of a static analysis with program transformations. We have implemented our method in the Glasgow Haskell Compiler and measured total memory allocations and execution times for some programs.

Keywords: lazy evaluation, Haskell, implementation of functional languages, recursive data structures

1. はじめに

遅延評価は，モジュール化を助け，プログラムの簡潔な記述を可能とするため，純関数型言語などで採用されている [1]．特にリストなどの再帰的データ構造を扱う際には，必要に応じて評価を進めることが可能となり，無限に続く

[1] 株式会社コマ・システムズ
Coma-Systems Co., Ltd., Minato, Tokyo 107–0052, Japan
[2] 電気通信大学大学院情報理工学研究科
Graduate School of Informatics and Engineering, The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[a)] takano@coma-systems.com

ようなデータを表現できる．

しかし，その一方で，計算の遅延に必要なオブジェクト（遅延オブジェクト，以下サンクと呼ぶ）をメモリ上に生成するための時間，空間が必要となることから，遅延評価にともなうコストは大きいという問題点がある．そのため，効率的に遅延評価を行う言語処理系においては，サンクの生成を抑制するために，たとえば，正格性解析 [2] や Cheapness Analysis [3] といった静的解析手法をこれまで採用してきた．正格性解析は必ず値が必要となるような式を静的に解析し，Cheapness Analysis は遅延させるまでもない軽い計算を解析する．それらの解析によって，遅延評価を前提としたプログラムの実行時間，使用メモリ領域ともに削減できることが確認されている．しかし，静的解析による手法は，解析の実装が複雑である上に，特定の文脈で用いられているサンクだけしか削減できないという問題点がある．

本論文では，リストのような線形に定義される再帰的データ構造に注目し，サンクの生成を抑える実装法を提案する．提案手法は，すでに割り当てられているサンクの保持している値を更新して再利用し，サンクの生成を抑える．これを可能にするため，コンパイル時にプログラム変換を行い，再利用するサンクへの参照が単一であるようにする．本手法を用いれば，たとえば，代表的な再帰的データ構造である線形リストでは，tail 部の遅延に必要となるサンクを再利用することができる．

本論文の構成は以下のとおりである．2 章では，まず，遅延評価における再帰的データ構造について説明し，既存の実装で必要となるサンクについて概観する．さらに，そのような実装には無駄があるという立場から，サンクを再利用する手法を提案する．3 章では，提案手法のフォーマルな定義と，必要となるプログラム変換に関して述べる．4 章では，Glasgow Haskell Compiler（GHC）のリストに提案手法を組み込む実装の詳細を述べる．5 章では，ベンチマークプログラムを用いた実験結果を報告する．6 章では，関連研究について述べる．7 章では，まとめと今後の課題について述べる．

提案手法は一般的な再帰的データ構造で必要となるサンクを削減することを可能とするが，線形リストを用いて説明する．また，本論文を通して，純関数型言語 Haskell [4] の記法に従い，プログラムなどを記述する．

2. 再帰的データ構造におけるサンクの再利用

遅延評価を行う言語においては，式の評価を遅延するためにサンクを生成する．ここでサンクとは，評価すべき式を記憶しておくためにヒープ中に割り当てる処理系内部のデータ構造であり，一般的には，遅延する式とその式を評価するのに必要な環境を内部に持つ．評価を遅延した式の値が実際に必要になった際には，サンクに記憶しておいた式を，やはりサンクに記憶しておいた環境において評価する．この操作をサンクの「強制」と呼ぶ．このような言語では，サンクのためのメモリ領域や強制のための操作が必要となるため，効率の良い実行は，先行評価を行う言語と比べ難しいといわれている．

本論文では，以下のような記法により，サンクを表現する．

$T_{label}\{exp\}\{env\}$

ここで，*exp* が遅延している式，*env* がその環境を示し，*label* によりそれぞれのサンクを区別する．また，文脈より明らかな場合には，$\{exp\}\{env\}$ の部分を省略する．

2.1 再帰的データ構造でのサンク

再帰的データ構造とは，線形リストや木などに代表されるように，その一部を取り出したとき，再びそのデータ構造が現れるようなデータ構造である．たとえば，各要素の型が a であるような線形リストは次のように定義することができる．

```
data List a = Nil | Cons a (List a)
```

ここで，データ構成子 Cons の第 2 引数が再帰的に List 型のデータを生成するため，後続も同様のデータ構造となり，線形に連なるデータ構造を表現している．以下簡潔のため，List a を [a] と，Nil を [] と，Cons を中置記法の : を用いて記述する．

遅延評価を行う言語においては，データ構造も遅延性を持ち，その構成要素はサンクとして必要になるまで評価されない．線形リストでいえばデータ構成子 Cons の引数は遅延させる計算の対象となる．たとえば，Haskell 標準ライブラリで定義されている関数 enumFromTo は以下のように定義されている．

```
enumFromTo :: Int -> Int -> [Int]
enumFromTo n m
  | m < n     = []
  | otherwise = n : enumFromTo (n+1) m
```

この関数は，引数に与えられた整数 n から m までを要素とするリストを生成するが，データ構成子 : がリストの後続となる式 enumFromTo (n+1) m を遅延するので，必要になるまでリストの後続の要素が生成されることはない．

ここで，既存の実装では，リストを読み進めるごとにサンクを生成する．たとえば，enumFromTo 1 10 という式により生成されるリストを読み進める計算を，サンクを明示して書くと次のようになる．

```
enumFromTo 1 10
  ⇒ 1:T1{enumFromTo (n+1) m}{n=1,m=10}
  ⇒ 1:2:T2{enumFromTo (n+1) m}{n=2,m=10}
  ⇒ 1:2:3:T3{enumFromTo (n+1) m}{n=3,m=10}
  ⇒ ···
```

まず，enumFromTo の定義に基づいて，$1{:}T_1$ というリスト

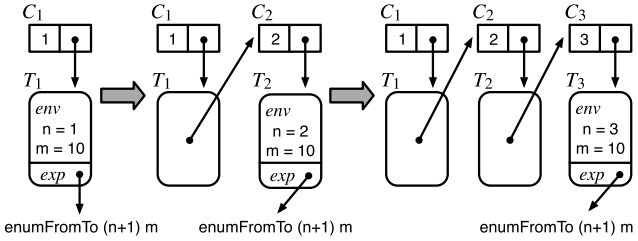


図 1　遅延線形リストの評価
Fig. 1　Evaluation process without thunk reuse.

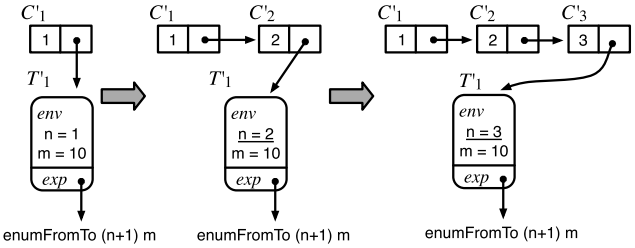


図 2　遅延線形リストにおけるサンク再利用
Fig. 2　Evaluation process with thunk reuse.

が構成される．このリストを進める際には，後続に指定されている T_1 を強制し，2:T_2 を得る．同様にして，後続のサンクを強制することで，遅延リストを読み進めることができる．

この様子を**図 1** に示す．ここで注目すべきは，リストを読み進めるごとに，T_1，T_2，T_3 といった新たなサンクをヒープ内に割り当てている点である．なお，call-by-need を実現するために，評価済みのサンクを，強制結果のオブジェクト（この例では C_2 や C_3）へのポインタ（間接ポインタ）で上書きする [5], [6]．このような処理過程では，後続を遅延するためのサンクがリストの長さに応じて必要となることが分かる．

2.2　提案手法

提案手法は，前節のような実装で，リストを読み進めるごとにサンクを新たに生成していたことに注目する．再帰的データ構造の後続の生成を遅延するためのサンクは同じような形をする傾向にある．実際 `enumFromTo` の例では，T_1，T_2，T_3 ともに `enumFromTo (n+1) m` という式を遅延するサンクであり，それらの違いは保持している環境だけである．このようなことから，提案手法は，再帰的データ構造の後続にあるサンクの保持している式や環境を更新し，再帰的データ構造の後続を構成する際に再利用し，サンクの生成を抑制する．

サンクの持つ式や環境を更新することにより矛盾が生じないようにするため，提案手法では，対象となるサンクへの参照がただ 1 つであるような場合しか再利用できない．そのため，提案手法は，コンパイル時のプログラム変換により，再利用対象のサンクがただ 1 つの参照によってしか指されないようにする．詳細については，2.3 節で述べる．なお，以降，本論文ではただ 1 つの参照で指されているサンクを，単一参照されているサンクと呼ぶ．また，図 1 に示したような既存の処理系で用いられている再利用されないサンクを「通常サンク」，本論文で提案する再利用するサンクを「再利用サンク」と呼ぶ．

前節で定義した関数 `enumFromTo` を用いて，提案手法を適用した場合の評価過程を見る．`enumFromTo 1 10` という式により生成されるリストを読み進める場合は，以下のようになる．

```
enumFromTo 1 10
  ⇒ 1:T1{enumFromTo (n+1) m}{n=1,m=10}
  ⇒ 1:2:T1{enumFromTo (n+1) m}{n=2,m=10}
  ⇒ 1:2:3:T1{enumFromTo (n+1) m}{n=3,m=10}
  ⇒ ···
```

この例では，サンクが保持している環境の中の `n` の値を更新することで，関数 `enumFromTo` の初回の呼び出し時に生成される T_1 を再利用している．提案手法では，リストを読み進めるごとに後続のサンクを生成しないので，前節の T_2，T_3 と次々にサンクを生成する場合と比べ，サンクの生成を抑えることができる．

サンクの再利用の様子を**図 2** に示す．再利用しない場合（図 1）は，サンクを間接ポインタで上書きすることによりリストを接続していたが，再利用する場合（図 2）は，リストを接続するためにコンスセルの tail 部を直接破壊的に書き換える必要がある．

この例においては，サンクの環境中の 1 つの変数 `n` の値だけを更新していたが，複数の変数の値を更新したり，遅延する式を更新したりすることもできる．

2.3　再利用するサンクの単一参照性

提案手法は，対象となるサンクが単一参照されていることに基づいて再利用を行う．このような前提を置くことで，サンクの再利用時に破壊的に書き換えるべき参照を，コンスセルの tail 部だけに限定することができる．ここでは，単一参照性について考える．

まず，リストを生成する関数について考える．たとえば，以下の関数 `f1` は，前節の `enumFromTo` と同様に再利用サンクを用いることができる．

```
f1 :: a -> [a]
f1 x = let xs = [x] in x:xs
```

`[x]` を遅延するサンクは関数 `f1` の返り値のコンスセルの tail 部からのみ参照されている．そのため，`[x]` を遅延するためには，サンクを再利用することができる．

これに対し，以下のようなリストを生成する関数 `f2` は，`[x]` を遅延するサンクへの参照を増やしてしまう．

```
f2 :: a -> ([a], [a])
f2 x = let xs = [x] in (x:xs, xs)
```

返り値のペアの第 1 要素と第 2 要素の間で，xs への参照が共有されてしまうこととなり，サンクへの参照を明らかに増やしてしまう．したがって，f2 では [x] を遅延するのに再利用可能なサンクを割り当てることはできない．

これらの例から分かるように，サンクを割り当てる際には，それを束縛する変数への参照が 1 カ所からであることを確認しなければならない．

次に，リストを参照するような関数で，サンクへの参照が増える場合について考える．GHC においては，パターンマッチングがデータ構造を分解し，その要素への参照を増やす．たとえば，以下の関数 g を考える．

```
g :: [a] -> (a,[a])
g (x:xs) = (x, xs)
```

関数 g はパターンマッチングで取り出された xs をペアの要素として取り出す．変数 xs に束縛されているサンクは，すでにコンスセル x:xs の tail 部から参照されているため，関数 g はこのサンクに対する参照を増やしてしまう．

そこで，本論文では，サンクに対する単一参照性を崩す恐れがある箇所に対する静的なプログラム変換を提供することで，つねに単一参照性が保たれるようにし，再利用を可能とする．ここでのプログラム変換は，次のようなものである．

> コンスセルの tail 部へのすべての参照（リストへのパターンマッチング）を，関数 tail# を使って，間接的な参照へと置換する．

関数 tail# は，リストのアクセサ関数 tail とほぼ同じである．異なるのは，間接ポインタを用いる代わりに，強制結果を tail# の引数のコンスセルの tail 部から直接接続するために，引数のコンスセルをスタックに保存しておく処理が追加されている点である．

たとえば，上で示した関数 g は以下のような関数 g’ に変換する．

```
g' :: [a] -> (a,[a])
g' xxs@(x:_) = (x, tail# xxs)
```

このような変換によりサンクに対する単一参照性を保つことができることは，関数 tail# が，リストの後続のサンクを返すのではなく，そのサンクを強制した値を返すということから分かる．あるリストに対して，関数 tail# が呼ばれたということは，弱頭部正規形（Weak Head Normal Form）まで簡約を行うことが要求されているので，tail# などの関数が後続のサンクそのものを強制せずに結果として返すことはない．また，サンクは第 1 級のオブジェクトではないため，プログラム中で明示的に後続のサンクを参照することはできない．つまり，関数 tail# のみを用いて，リストの後続として指定されている再利用サンクを直に参照することは不可能である．

上の例で，tail 部の再利用サンクが直接指されることがないことは，**図 3** のように，すべての参照がコンスセルを通して行われるようになることから分かる．プログラム変換前では，O_1 や O_2 が RT_1 を直接参照しているが，変換後では，O_1 や O_2 は T_t からの間接参照になっている．この図では，tail# xxs のためのサンク T_t ができているが，その式が正格な文脈であれば，このようなことにはならない．

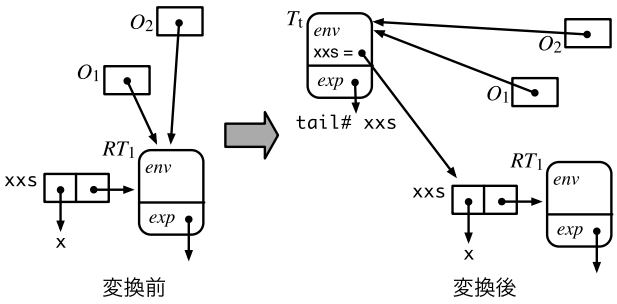


図 3　tail# による間接参照
Fig. 3　Indirection reference using tail#.

3. 提案手法

前章では，提案手法の動作を概観した．提案手法のプログラム変換は，

$\mathcal{C}$：パターンマッチングにおける tail# の導入
$\mathcal{P}$：再利用するサンクの明示された式への変換

の 2 つからなる．はじめに変換 $\mathcal{C}$ を行って，その後に変換 $\mathcal{P}$ を行う．変換 $\mathcal{C}$ は，2.3 節における g から g’ への変換，すなわちコンスセルのパターンマッチングにおける tail 部の変数の使用を tail# の呼び出しに変換する．また，変換 $\mathcal{P}$ では，どのサンクを再利用サンクにすることができるかを解析し，特定する．なお，2.3 節で説明した，リストを生成する側での単一参照性については，変換 $\mathcal{P}$ で考慮する必要がある．

本章では，これらのプログラム変換のフォーマルな定義を示す．

3.1 文法

両変換で共通に用いる言語の文法を，**図 4** に示す．データ構造はリストに限定しているので，データ構成子は Nil と Cons だけである．

$Expr^-$ は入力として与えられる式で，GHC の内部言語として使われている Core 言語の古い形式の文法 [7] に似ている．$Expr^-$ の特徴は，関数適用と Cons の引数が変数に限定されている点である．そのため，ヒープへのサンクの割当ては，let 式における局所変数への束縛値の式を遅延させるときだけに限定される．ここでのサンクは，まだ通常サンクである．また，サンクは case 式の e^- の箇所において強制され，その値が x に束縛されたうえでパターン p^- の評価へ移る．データ構造はリストだけなので，パターン

Expressions（入力）

$$
\begin{aligned}
& x \in \mathit{Var} \\
& e^- \in \mathit{Expr}^- ::= \backslash x \to e^- \mid \mathsf{Nil} \mid \mathsf{Cons}\, x_h\, x_t \mid x \mid e^-\, x \\
& \qquad \mid \mathsf{case}\, e^-\, \mathsf{of}\, x\, p^- \mid \mathsf{let}\, x = e_1^-\, \mathsf{in}\, e_2^- \\
& p^- \in \mathit{Pattern}^- ::= \{\, \mathsf{Nil} \to e_n^-;\ \mathsf{Cons}\, x_h\, x_t \to e_c^- \,\}
\end{aligned}
$$

Expressions（出力）

$$
\begin{aligned}
& x \in \mathit{Var} \\
& e \in \mathit{Expr} ::= \backslash x \to e \mid \mathsf{Nil} \mid \mathsf{Cons}\, x_h\, x_t \mid x \mid e\, x \mid \mathsf{case}\, e\, \mathsf{of}\, x\, p \\
& \qquad \mid \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \mid \mathsf{letreuse}\, x = e_1\, \mathsf{in}\, e_2 \mid \mathsf{tail\#}\, x \\
& p \in \mathit{Pattern} ::= \{\, \mathsf{Nil} \to e_n;\ \mathsf{Cons}\, x_h\, x_t \to e_c \,\}
\end{aligned}
$$

図 4　文法
Fig. 4　Syntax.

では Nil の場合と Cons の場合の分岐を行う．

Expr は変換 $\mathcal{C}$ の出力，および変換 $\mathcal{P}$ の入出力であり，Expr^- に tail# 式と letreuse 式を追加したものである．letreuse は再利用対象のサンクを明示する構文である．これは let に似ているが，e_1 の計算を遅延させるのに再利用サンクが未生成のときは新たに生成し，再利用サンクがすでにある場合はそれを再利用する点が let と異なる．

3.2　プログラム変換 $\mathcal{C}$

変換 $\mathcal{C}$ は，プログラム中の再利用サンクを直接参照しうるすべての式を，間接参照へ置き換える．この変換により，提案手法で必要としている再利用サンクの単一参照性を成り立たせることができる．

図 5 に変換 $\mathcal{C}$ を示す．変換 $\mathcal{C}$ は，2.3 節で説明したように，パターンマッチにより分解されたコンスセルの tail 部に相当する変数を，そのコンスセルに対する tail# に置換する．

変換 $\mathcal{C}$ は，変換対象の Expr^- と *Map* を受け取り，*Expr* を返す．ここで，データ構造 *Map* は，$\langle x_t, x\rangle$ というような変数の組の集合で，x_t を $\mathsf{tail\#}\, x$ で置き換えるという情報を保持するのに用いる．*Map* への変数の組の追加は，case 式で Cons のパターンを取り出す際に行う．また，変数の tail# 式への置換は，$\mathcal{C}[\![\mathsf{Cons}\, x_h\, x_t]\!]$ と $\mathcal{C}[\![x_t]\!]$ と $\mathcal{C}[\![e^-\, x_t]\!]$ において行う．いずれの場合も x_t が *Map* に含まれていれば，新たな局所変数を導入して tail# へと置き換える．なお，この置換には let 式によって新たに $\mathsf{tail\#}\, x$ を遅延するサンクを割り当てる可能性があることに注意されたい．ただし，$\mathsf{tail\#}\, x$ が正格な文脈に出現していれば，このようなサンクを割り当てる必要はない．

3.3　プログラム変換 $\mathcal{P}$

変換 $\mathcal{P}$ は，let 式で束縛される変数について，束縛値に再利用サンクを使うことができれば letreuse に置換する．ここで，let の局所変数 x が次の条件をすべて満足する場合に，x の束縛値のサンクは再利用可能であると判断する．

$$
\begin{aligned}
M \in \mathit{Map} &::= \phi \mid M, \langle x_t, x\rangle \\
\mathcal{C} &:: \mathit{Expr}^- \to \mathit{Map} \to \mathit{Expr} \\
\mathcal{C}[\![\backslash x \to e^-]\!]\, M &= \backslash x \to \mathcal{C}[\![e^-]\!]\, M \\
\mathcal{C}[\![\mathsf{Nil}]\!]\, M &= \mathsf{Nil} \\
\mathcal{C}[\![\mathsf{Cons}\, x_h\, x_t]\!]\ (M, \langle x_h, x_1\rangle, \langle x_t, x_2\rangle) &= \mathsf{let}\, x_1' = \mathsf{tail\#}\, x_1\ \mathsf{in}\ \mathsf{let}\, x_2' = \mathsf{tail\#}\, x_2\ \mathsf{in}\ \mathsf{Cons}\, x_1'\, x_2'\ \mathsf{where}\ x_1' \text{ and } x_2' \text{ are fresh variables} \\
\mathcal{C}[\![\mathsf{Cons}\, x_h\, x_t]\!]\, (M, \langle x_h, x\rangle) &= \mathsf{let}\, x' = \mathsf{tail\#}\, x\ \mathsf{in}\ \mathsf{Cons}\, x'\, x_t\ \mathsf{where}\ x' \text{ is a fresh variable} \\
\mathcal{C}[\![\mathsf{Cons}\, x_h\, x_t]\!]\, (M, \langle x_t, x\rangle) &= \mathsf{let}\, x' = \mathsf{tail\#}\, x\ \mathsf{in}\ \mathsf{Cons}\, x_h\, x'\ \mathsf{where}\ x' \text{ is a fresh variable} \\
\mathcal{C}[\![\mathsf{Cons}\, x_h\, x_t]\!]\, M &= \mathsf{Cons}\, x_h\, x_t \\
\mathcal{C}[\![x_t]\!]\, (M, \langle x_t, x\rangle) &= \mathsf{tail\#}\, x \\
\mathcal{C}[\![x]\!]\, M &= x \\
\mathcal{C}[\![e^-\, x_t]\!]\, (M, \langle x_t, x\rangle) &= \mathsf{let}\, x' = \mathsf{tail\#}\, x\ \mathsf{in}\ (\mathcal{C}[\![e^-]\!]\, M)\, x'\ \mathsf{where}\ x' \text{ is a fresh variable} \\
\mathcal{C}[\![e^-\, x]\!]\, M &= (\mathcal{C}[\![e^-]\!]\, M)\, (\mathcal{C}[\![x]\!]\, M) \\
\mathcal{C}[\![\mathsf{case}\, e^-\, \mathsf{of}\, x\, \{\ \mathsf{Nil} \to e_n^-;\ \mathsf{Cons}\, x_h\, x_t \to e_c^-\ \}]\!]\, M &= \mathsf{case}\, (\mathcal{C}[\![e^-]\!]\, M)\, \mathsf{of}\, x\, \{\ \mathsf{Nil} \to \mathcal{C}[\![e_n^-]\!]\, M;\ \mathsf{Cons}\, x_h\, x_t \to \mathcal{C}[\![e_c^-]\!]\, (M, \langle x_t, x\rangle)\ \} \\
\mathcal{C}[\![\mathsf{let}\, x = e_1^-\, \mathsf{in}\, e_2^-]\!]\, M &= \mathsf{let}\, x = \mathcal{C}[\![e_1^-]\!]\, M\, \mathsf{in}\, \mathcal{C}[\![e_2^-]\!]\, M
\end{aligned}
$$

図 5　プログラム変換 $\mathcal{C}$
Fig. 5　Transformation $\mathcal{C}$.

（**条件 1**）　let 式中で，x の出現回数が 1 回である．
（**条件 2**）　x が let の値として返されるリストの背骨を構成する Cons の tail 部（第二引数）に使われている．

ここでいうリストの背骨とは，tail 部で接続されている一連のコンスセルをいう．

図 6 に変換 $\mathcal{P}$ を示す．前節で述べたように，変換 $\mathcal{C}$ を適用済みのプログラムを変換 $\mathcal{P}$ の入力とするので，$\mathcal{P}$ は *Expr* から *Expr* への変換である．let 式について，letreuse に置換できるかどうかを，条件 1 について関数 *occ* で解析し，条件 2 について関数 *spine* で解析する．*occ* と *spine* の定義を図 7 に示す．まず，*occ* は，式と変数を受け取って，その式中に指定された変数があるかどうかを判定する関数である．変数 x が Cons の第 2 引数として与えられることだけが許容されていることに注意されたい．また，*spine* は，式と変数を受け取って，指定された変数が Cons の第 2 引数に出現するかどうかを判定する関数である．ここで，letreuse が入れ子となっていないことも同時に確か

$$
\begin{aligned}
\mathcal{P} &:: \mathit{Expr} \to \mathit{Expr} \\
\mathcal{P}[\![\backslash x \to e]\!] &= \backslash x \to \mathcal{P}[\![e]\!] \\
\mathcal{P}[\![\mathsf{Nil}]\!] &= \mathsf{Nil} \\
\mathcal{P}[\![\mathsf{Cons}\ x_1\ x_2]\!] &= \mathsf{Cons}\ x_1\ x_2 \\
\mathcal{P}[\![x]\!] &= x \\
\mathcal{P}[\![e\ x]\!] &= \mathcal{P}[\![e]\!]\ x \\
\mathcal{P}[\![\mathsf{case}\ e\ \mathsf{of}\ x\ \{ & \\
\quad \mathsf{Nil} \to e_n; & = \\
\quad \mathsf{Cons}\ x_h\ x_t \to e_c\ \}]\!] & \\
\mathcal{P}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] &= \mathsf{if}\ \mathit{spine}\,\mathcal{P}[\![e_2]\!]\ x = \mathit{Once} \\
& \quad \&\&\ \mathit{occ}\ e_1\ x = \mathsf{True} \\
& \quad \&\&\ \mathit{occ}\ e_2\ x = \mathsf{True} \\
& \quad \mathsf{then}\ \mathsf{letreuse}\ x = \mathcal{P}[\![e_1]\!]\ \mathsf{in}\ \mathcal{P}[\![e_2]\!] \\
& \quad \mathsf{else}\ \mathsf{let}\ x = \mathcal{P}[\![e_1]\!]\ \mathsf{in}\ \mathcal{P}[\![e_2]\!] \\
\mathcal{P}[\![\mathsf{letreuse}\ x = e_1\ \mathsf{in}\ e_2]\!] &= \mathsf{letreuse}\ x = e_1\ \mathsf{in}\ e_2 \\
\mathcal{P}[\![\mathsf{tail\#}\ x]\!] &= \mathsf{tail\#}\ x
\end{aligned}
$$

図 6　プログラム変換 $\mathcal{P}$
Fig. 6　Transformation $\mathcal{P}$.

$$
\begin{aligned}
\mathit{occ} &:: \mathit{Expr} \to \mathit{Var} \to \mathit{Bool} \\
\mathit{occ}\ (\backslash x_1 \to e)\ x &= \mathit{occ}\ e\ x \\
\mathit{occ}\ \mathsf{Nil}\ x &= \mathsf{True} \\
\mathit{occ}\ (\mathsf{Cons}\ x_1\ x_2)\ x &= x_1 \neq x \\
\mathit{occ}\ x_1\ x &= x_1 \neq x \\
\mathit{occ}\ (e\ x_1)\ x &= \mathit{occ}\ e\ \&\&\ x_1 \neq x \\
\mathit{occ}\ (\mathsf{case}\ e\ \mathsf{of}\ x_s\ \{ & \quad \mathit{occ}\ e\ x \\
\quad \mathsf{Nil} \to e_n; & = \quad \&\&\ \mathit{occ}\ e_n\ x \\
\quad \mathsf{Cons}\ x_h\ x_t \to e_c\ \})\ x & \quad \&\&\ \mathit{occ}\ e_c\ x \\
\mathit{occ}\ (\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2)\ x &= \mathit{occ}\ e_1\ x\ \&\&\ \mathit{occ}\ e_2\ x \\
\mathit{occ}\ (\mathsf{letreuse}\ x_1 = e_1\ \mathsf{in}\ e_2)\ x &= \mathit{occ}\ e_1\ x\ \&\&\ \mathit{occ}\ e_2\ x \\
\mathit{occ}\ (\mathsf{tail\#}\ x_1)\ x &= x_1 \neq x \\
\\
\mathit{Reuse} &= \mathit{None} \mid \mathit{Once} \mid \mathit{NoMore} \\
\mathit{spine} &:: \mathit{Expr} \to \mathit{Var} \to \mathit{Reuse} \\
\mathit{spine}\ (\backslash x_1 \to e)\ x &= \mathit{None} \\
\mathit{spine}\ \mathsf{Nil}\ x &= \mathit{None} \\
\mathit{spine}\ (\mathsf{Cons}\ x_1\ x_2)\ x &= \mathsf{if}\ x_2 = x\ \mathsf{then}\ \mathit{Once} \\
& \qquad\qquad \mathsf{else}\ \mathit{None} \\
\mathit{spine}\ x_1\ x &= \mathit{None} \\
\mathit{spine}\ (e\ x_1)\ x &= \mathit{None} \\
\mathit{spine}\ (\mathsf{case}\ e\ \mathsf{of}\ x_s\ \{ & \\
\quad \mathsf{Nil} \to e_n; & = \mathit{spine}\ e_n\ x \uparrow \mathit{spine}\ e_c\ x \\
\quad \mathsf{Cons}\ x_h\ x_t \to e_c\})\ x & \\
\mathit{spine}\ (\mathsf{let}\ x_1 = e_1\ \mathsf{in}\ e_2)\ x &= \mathit{spine}\ e_2\ x \\
\mathit{spine}\ (\mathsf{letreuse}\ x_1 = e_1\ \mathsf{in}\ e_2)\ x &= \mathit{NoMore} \\
\mathit{spine}\ (\mathsf{tail\#}\ x_1)\ x &= \mathit{None}
\end{aligned}
$$

図 7　変換 $\mathcal{P}$ のための補助関数
Fig. 7　Helper functions.

め，入れ子となっていた場合は *NoMore* を返す．ここで，$\uparrow$ は $\mathit{None} < \mathit{Once} < \mathit{NoMore}$ という順序で値を比較し，大きいほうを返す関数である．

4. 実装

純関数型言語 Haskell の代表的な処理系である Glasgow Haskell Compiler（GHC）[8] のバージョン 7.0.3 に，提案手法を実装した．

4.1 概要

GHC は**図 8** に示すように，3 種類の中間言語を持つ．

Core 言語は，構文糖衣が除かれた Haskell 言語に近く，3.1 節で用いた言語のもとになったものである．また，STG 言語は，遅延評価機構 Spineless Tagless G-machine（STG）[6] の動作モデルに合わせた言語であり，サンクが明示されている．これら 2 つの言語において，GHC は正格性解析や let-floating [9] などの最適化を行う．最後に，その最適化がなされたコードを C−− 言語を経由してターゲットコードへと出力する．

提案手法は，コンパイラと実行時システムに変更を加えることで実装した．コンパイラへは，$\mathcal{C}$ と $\mathcal{P}$ の変換を追加し，コード生成系にも修正を加えた．まず，変換 $\mathcal{C}$ については，Core-to-Core の変換パスで実装した．Core 言語では Haskell 言語の変数名が残っているため，`tail#` の呼び出しに変換するのに適している．次に，変換 $\mathcal{P}$ は STG-to-STG のパスで適用するようにした．STG 言語は，サンクが明示された形の言語であるため，letreuse への置換が容易であるためである．また，STG 言語から C−− のコード生成時において，letreuse に対するコードを生成するための変更に加えて，コンスセル（データコンストラクタ）に関するコード生成にも修正が必要であった．

現在は，変換 $\mathcal{C}$ と変換 $\mathcal{P}$ は図 5，6，7 で示したものをほぼそのまま実装しており，*occ* などの補助関数が同じ式に対して繰り返し呼ばれる可能性がある．ここで，コンパイル時間を考慮すれば，式ごとの対応表を用意しておくほうが望ましい．しかし，これらの変換に対してかかる時間

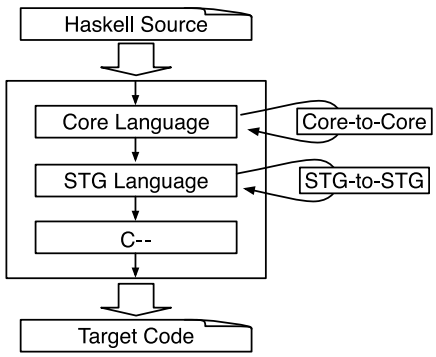


図 8　GHC のコンパイルパス
Fig. 8　Compile path of GHC.

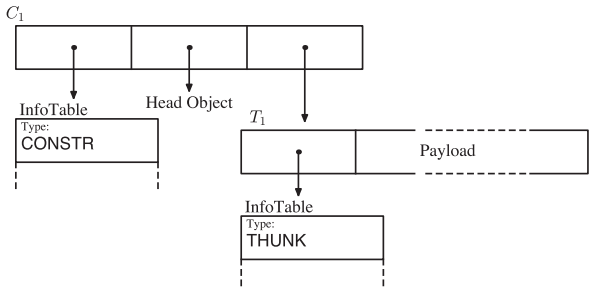


図 9　GHC 上でのコンスセルとサンクのデータ構造
Fig. 9　Representation of cons cell and thunk in GHC.

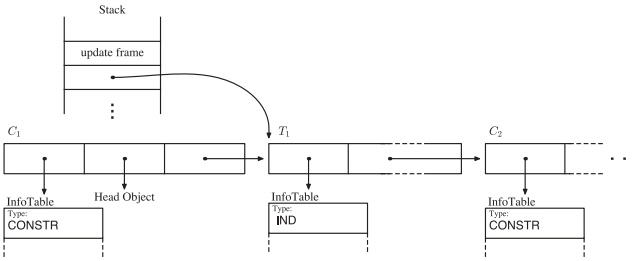


図 10　GHC 上での update frame によるサンクの上書き
Fig. 10　Updating using update frame.

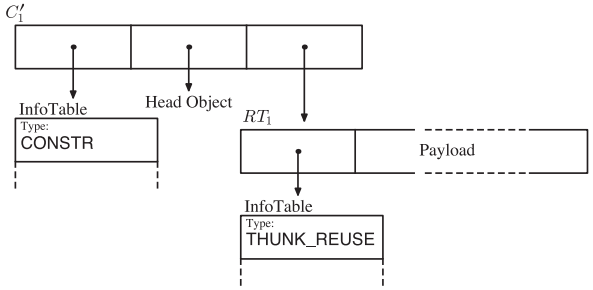


図 11　提案手法を導入済みの GHC 上でのコンスセルと再利用するサンクのデータ構造
Fig. 11　Representation of cons cell and reusable thunk in our implementation.

は，総コンパイル時間に対して小さいため，現状では対応表を用意せずに実装することとした．なお，`nofib` ベンチマーク [10] の real サブセットにある anna というプログラム（全 31 ファイル，9520 行）のコンパイル時間は，16.4 秒に対して 18.7 秒と，約 14.1%の増加であった．

実行時システムには，再利用サンク用のオブジェクトタイプとコードを追加し，さらに再利用サンクの更新を行う再利用機構を追加した．また，ごみ集めも，再利用サンクに対応するように変更した．

なお，関数 `map` や `filter` などの Haskell の基本（Prelude）関数については，提案手法のコンパイラでコンパイルし，置き換える必要がある．

4.2　再利用サンクの実現

本節では，まず既存の GHC が通常サンクをどのように扱っているかを説明し，その後，再利用の詳細を説明する．

既存の GHC は，**図 9** のようにしてデータ構造と通常サンクを表現している．GHC 中のオブジェクトは，オブジェクトのタイプ，コード，ごみ集め時の情報などが入った *InfoTable* を先頭に保持し，その後に（あれば）必要な非ポインタ，ポインタを並べて保持する．以後，これらの非ポインタおよびポインタのデータ領域をペイロードと呼ぶ．コンスセルは，ペイロードに head 部と tail 部へのポインタを持つ．また，通常サンクの場合には，環境中の各要素へのポインタが並んでいる．そのサンクが遅延する式のコンパイル済みコードへの参照は，InfoTable 中に保持する．

このような通常サンクの強制の手順は，以下のとおりである．

(1) *update frame* と呼ばれるフレームとともに，強制する通常サンクをスタックに積む．
(2) 通常サンクの保持している環境のもとで，そのサンクが遅延している式を評価し，レジスタと呼ばれる GHC 上の決まった領域に結果の値を書き込む．
(3) スタックトップの update frame を実行する．
(4) update frame では，スタックに積んである強制済みの通常サンクを書き換え，レジスタにある強制結果への参照を持つ間接ポインタとする．

図 9 の通常サンクについて，強制後のスタックとオブジェクトの様子を**図 10** に示す．update frame は，スタックに積んである強制済みサンクへの参照を利用して，図 9 の通常サンク T_1 のオブジェクトタイプを間接参照ポインタを表す IND へと変更し，さらに，ペイロードが強制結果を参照するよう書き換える．このようにして，強制したサンクを参照しているオブジェクト間で強制結果を共有する．

提案手法では，コンスセルの tail 部から指される再利用サンクを強制中に，そのコンスセルの後続のコンスセルの tail 部を遅延させるための新たなサンクを生成するかわりに再利用する．ただし，再利用サンクを間接参照オブジェクトに上書きすることができないので，再利用サンクの参照元が強制結果を直接参照するようにしなければならない．ここで，再利用サンクはコンスセルの tail 部から単一参照されているので，直接参照するように書き換えるのはその tail 部だけである．

以上のような方針に基づいて，再利用サンクを表現するオブジェクトタイプを新たに追加し，**図 11** のようにした．オブジェクトタイプが THUNK_REUSE となっていること以外は，通常サンクと同様である．

これに対し，以下のような手順でサンクを再利用する

(1) 再利用サンクの強制処理に入る前に，関数 `tail#` は *reuse frame* というフレームとともに，再利用サンクへの参照を持つコンスセル（`tail#` の引数）をスタックに積む．変換 $\mathcal{C}$ により，再利用対象のサンクが強

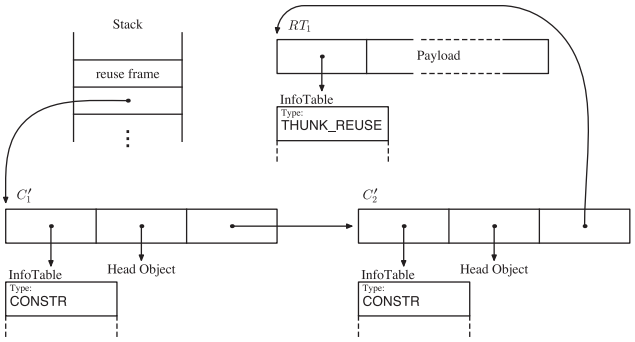


図 12　提案手法を導入済みの GHC 上でのサンクの評価
Fig. 12　Thunk reuse with the proposed method.

制される直前には `tail#` が呼ばれることに注意されたい．

(2) 再利用サンクの保持している環境のもとで，そのサンクが遅延している式を評価し，レジスタに結果の値を書き込む．強制中に必要であれば（結果の値がコンスセルの場合），再利用サンクの式・環境を更新して用いる．

(3) スタックトップの reuse frame を実行する．

(4) reuse frame では，レジスタの内容をもとに，サンクの参照元であるコンスセル（スタックに積まれている）の tail 部を，レジスタにある結果の値を直接指すように破壊的に書き換える．

図 11 のサンクについて，この操作で評価した後のスタックとオブジェクトの様子を，**図 12** に示す．reuse frame 実行前には，C'_1 の tail 部は RT_1 を参照しているが，`tail#` が reuse frame とともに C'_1 へのポインタを積んでおくことで，C'_2 への参照に書き換えることができる．

この方法では，(4) における reuse frame による書き換えにより，間接参照オブジェクトを通さずに強制済みサンクの値を参照することができるようになる．そのため，従来の GHC と比べて，評価済みサンクの参照コストを小さくできる．ただし，間接ポインタはごみ集めによって直接参照へと書き換えられるので，このような優位性を得られるのは，次回のごみ集めが起こるまでに限定される．

続いて，以下のような式を例に，サンクの再利用処理に対して，どのようなコードがコンパイラより生成されるかを擬似コードを用いて説明する．

$f\ x\ y = \mathsf{letreuse}\ x_t = g\ x\ y\ \mathsf{in}\ \mathsf{Cons}\ x\ x_t$

ここで，x_t への束縛値は再利用サンクであり，自由変数の x と y はそのサンクの環境として保持される．f が呼び出される文脈として考慮すべきは，以下の場合である．

(1) 通常サンクによって，$f\ x\ y$ が遅延されている場合

(2) 再利用サンクによって，$f\ x\ y$ が遅延されている場合

(3) `main` 関数から正格な文脈で呼ばれている場合

まず，(1) の場合は，たとえば

$\mathsf{let}\ z = f\ x\ y\ \mathsf{in}\ \ldots\ z\ \ldots$

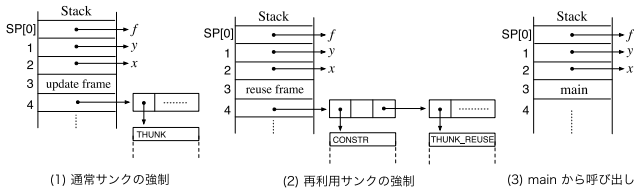


図 13　擬似コードの実行前のスタック
Fig. 13　Stack before pseudocode is executed.

のようにして f が呼ばれており，再利用サンクはまだ割り当てられていない．この通常サンクが強制されれば，**図 13** (1) のように update frame がスタックへと積まれている．

次に，(2) の場合には，すでに再利用サンクがヒープ上に確保されている．たとえば，以下のような y_t に束縛されている再利用サンクが強制中ということが考えられる．

$\mathsf{letreuse}\ y_t = f\ x\ y\ \mathsf{in}\ \mathsf{Cons}\ x\ y_t$

この例では，強制中の再利用サンクを $g\ x\ y$ のために再利用する．図 13 (2) に，擬似コードに入る直前のスタックを示す．前節で述べたように，この f の呼び出し後には `tail#` によって積まれた reuse frame が実行されるようになっている．

また，(3) の場合にも，(1) と同様にまだ再利用可能なサンクが割り当てられていない．

以上のことをふまえて，f に対する擬似コードは，**図 14** のようになる．まず，`HP+=3` によって，コンスセルのための領域をヒープ上に確保する．次に，現在強制中のサンクが再利用可能であるかチェックする．再利用可能である条件は，以下のとおりである．

- 再利用サンクがヒープに割当て済みである．
- 再利用サンクがすでにある場合には，そのサンクに今回再利用しようとするだけの十分な領域がある．サンクのサイズは自由変数の数であることに注意されたい．

再利用可能であれば，そのサンクが保持している値を更新する．擬似コード中では，`rt` で再利用サンクを参照している．また，可能でなければ，`HP+=3` として，再利用サンクをヒープに新たに割り当てる．最後に，コンスセルの tail 部に再利用サンクを設定する．

4.3　ごみ集め

GHC は世代別コピー型のごみ集めを採用している．

世代別ごみ集めは，大部分のオブジェクトは使用後にすぐ破棄され，長く生存しつづけるオブジェクトは生存期間が長いという経験則に着目した方式である．オブジェクトの生成時期に応じた世代（新世代と旧世代）に分割し，世代ごとにメモリの回収を行うことで，ごみ集めの効率化を図る．世代ごとにごみ集めを行うので，新世代への参照を行っている旧世代オブジェクトに関して，remebmered set と呼ばれるリスト（GHC では mutable list と呼ぶ）によって特別に扱わなくてはならない．新世代領域は 2 つに

```
// SP: stack pointer
// HP: heap pointer
HP += 3;  // Allocate memory for cons cell
x = SP[2];
y = SP[1];

if (SP[3] != &reuse_frame_info) {
goto ALLOCATE;
}
rt = SP[4].payload[1]; // reusable thunk
SP -= 3;               // pop x and y
if (rt.payload_size < 2) {
goto ALLOCATE; // rt is too small
}

OVERWRITE:
rt[0] = info table for g x y
rt[1] = x;
rt[2] = y;
tailobj = &rt[0];
goto DONE;

ALLOCATE:
// allocate memory for reusable thunk
HP += 3;
HP[-5] = info table for g x y
HP[-4] = x;
HP[-3] = y;
tailobj = &HP[-5];

DONE:
// building up cons cell
HP[-2] = info table of cons cell;
HP[-1] = x;
HP[0]  = tailobj;
```

図 14　letreuse 式のコンパイル済みコード
Fig. 14　Pseudocode for letreuse expression.

分け，ごみ集めの際に生存しているオブジェクトのみをコピーする．

提案手法を実現する際には，再利用サンクはだんだん古くなっていくことを考慮しなくてはならない．また，環境の更新により，再利用サンクがペイロードに保持する参照がそのサンクよりも新しい領域になりうるという点も，通常サンクと異なる点である．

このような通常サンクとの違いを解消するために，再利用サンクをつねに新世代へ置くこととする．GHC は，ごみ集めの対象となる世代より若い世代についてコピー方式ごみ集めを行うため，再利用するサンクがごみ集めの対象となった際に，再利用サンクのコピー先の世代を調整することができる．

5. 実験

提案手法を GHC 7.0.3 に変更を加えて実現し，効果を確認した．実験には，nofib ベンチマーク [10] の real サブセットのプログラムを用いた．前述したように，Haskell の基本関数は提案手法を用いてコンパイルしたものに置き換えた．また，コンパイル時には，-O2 オプションを付けており，GHC の正格性解析を行ったものとなっている．実験は AMD Opteron CPU（2.3 GHz），メインメモリ 8 GB の PC 上で動作している Linux kernel 2.6.32 上で行い，GHC の実行時オプション -S によって，総メモリ割当て量と実行時間を測定し，元の GHC に対する削減率とオーバヘッドを計算した．

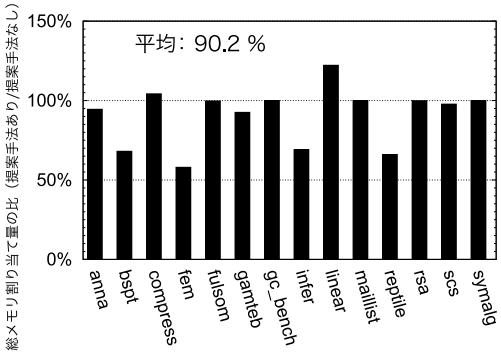


図 15　実験結果：総メモリ割当て量
Fig. 15　Total memory allocation relative to original GHC.

5.1 総メモリ割当て量

まず，総メモリ割当て量の結果を図 **15** に示す．縦軸に提案手法なしの場合を 100 としたときの比をとっている．平均は，90.2%であった．これは，正格性解析で削減できなかったサンクの生成を削減できていることを意味している．

ベンチマーク別に見ると，fem や reptile などのベンチマークにおいて大幅な改善が見られる．これらのベンチマークは，リストの消費が多いプログラムであるためと考えられる．逆に linear については 122.2%と提案手法を適用した場合の総メモリ割当て量が増加してしまっている．これは，3.2 節のプログラム変換 $\mathcal{P}$ において導入される tail# 式に対するサンクを，正格性解析により除去できなかったためと考えられる．

5.2 実行時間

実行時間の結果を図 **16** に示す．縦軸に提案手法なしの場合を 100 としたときの比をとっている．平均は，107.2%であった．reptile や linear などのベンチマークでは，大きなオーバヘッドがかかる結果となってしまっている．

提案手法は，メモリ割当て時間を削減でき，ごみ集めの回数も減らすことができる可能性がある．しかし，その一方で，図 14 で示したような実行時のチェックが必要である．GHC のメモリ割当てやごみ集め時間が元々 Haskell の実行モデルに合わせてチューニングされ，高速であるため，現状では遅くなってしまうという結果となった．実行時チェックによるオーバヘッドの削減は今後の課題である．

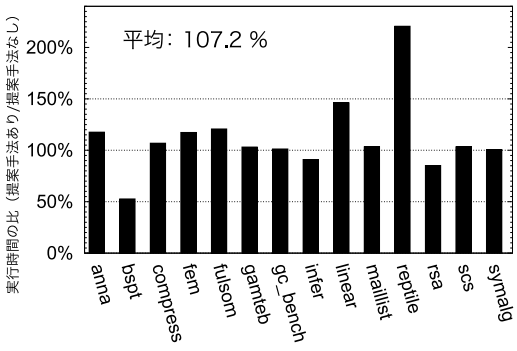


図 16　実験結果：実行時間
Fig. 16　Execution time relative to original GHC.

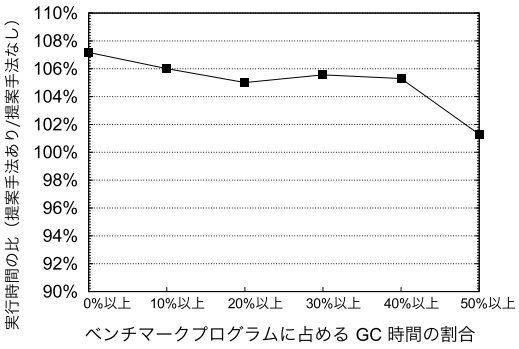


図 17　実験結果：ごみ集め時間と実行時間比
Fig. 17　GC time and execution time.

次に，全体に占めるごみ集め時間の割合ごとに，プログラムを分類し，実行時間のオーバヘッドを集計したものを**図 17** に示す．縦軸に提案手法なしの場合を 100 としたときの実行時間の比，横軸にごみ集め時間の割合ごとにベンチマークプログラムを分類した場合の値をとっている．若干ではあるが，ごみ集め時間の割合が増えるに従って，プログラムの実行時間も本手法なしの場合に近づいている．この結果から，全実行時間に占めるごみ集めの割合が大きいプログラムに対しては，依然としてオーバヘッドの方が大きいものの，本手法による高速化の効果が大きくなる傾向にあることが分かる．

6. 関連研究

STG [6] には，update in place という手法が提案されている．これは，強制中のサンクを強制結果のために書き換えて使用することで，メモリ割当て量の削減を試みるものである．たとえば，5 つのペイロードを持つようなサンクを強制し，2 つのペイロードが必要なコンスセルができる場合には，5 つのうち 2 つのペイロードを使って，コンスセルとする．一方で，提案手法では強制中のサンクを，強制により作られるコンスセルの後続のサンクとして再利用する．2.2 節で見たように，後続のサンクは強制中のサンクと同様の形をとる傾向があるといえるので，提案手法のほうが，効率良く再利用できる可能性がある．

関数型言語において，データ構造の破壊的な書き換えによってプログラムの効率を上げるという研究はさかんに行われている．Concurrent Clean [11] は uniqueness typing [12] を導入することで，データ構造の破壊的書き換えを可能としている．ユーザは，破壊的な書き換えのための定義をし，Clean のコンパイラが正しく書き換えることができるかどうかをチェックする．ユーザの記述によっているという点で，提案手法がプログラム変換によって再利用サンクの単一参照性を保っていることと異なる．また，データ構造の書き換えに関する別の研究として，Hage らの研究 [13] がある．その研究では，文法を少し拡張することで，書き換え可能かどうかの情報を型システムで解析できるようにしている．サンクは処理系で内部的に作られるオブジェクトであるため，本研究はこれらの研究で対象としているデータ構造の破壊的な書き換えとは異なる．しかし，Hage らの用いている解析手法は，提案手法でのプログラム変換に取り入れることができる可能性がある．

サンクの単一参照性に注目した遅延評価処理系の効率化に関する研究には，update avoidance [14] がある．これは，単一参照であるサンクを強制結果で上書きする操作は，それ以降参照されないことを考えれば無駄であることに着目し，上書きの必要のないサンクを静的に解析している．本研究は，上書きの操作によるオーバヘッドでなく，サンクに必要となる無駄なメモリ領域を削減することを目的としているため，目的，手法ともに大きく異なる．

正格性解析 [2] や Cheapness Analysis [3] などの静的解析手法では，遅延させる必要がない式を静的に解析し，不要なサンクを削減する．これらの解析では，あるデータ構造に関して，その値が使用される文脈に応じて，ある構成要素まで正格であるといった判定が可能である．たとえば，リストの長さを求める `length` に与えられるリストは最後までたどることが明らかであるため，最後尾まで正格であると判定する．複雑な解析が必要となる場合が多いが，非常に強力なサンク削減手法である．しかし，文脈によって正格であると判断できない場合や無限データ構造に関しては，サンクを生成せざるをえない．そのため，上の `length` の例では，既存の静的解析手法においては，通常長さに比例する数のサンクが必要となる．それに対して提案手法では，再帰的なデータ構造であれば，サンクの生成を抑えて実行できる．つまり，提案手法とこれらの解析手法は削減の対象としているサンクが別であり，あわせて用いることが可能である．

Optimistic Evaluation [15] における Chunky Evaluation は遅延データ構造をある長さ分だけ投機的に進めることで，サンクを削除するという手法である．ある区間のみ正格に評価を進めることで，遅延評価プログラムの利点を残しつつ，サンクの生成などのオーバヘッドの削減を目指している．ただし，投機的にリストを進めるので，投機に失敗した場合には，無駄なリストを生成する可能性がある．

それに対して，提案手法では，実装面からのアプローチにより，遅延データ構造を保ちつつサンクの削減を目指している．そのため，プログラムの動作は変更せず，必要以上にリストを進めることはない．一方，提案手法と Chunky Evaluation とは直交する手法であるので，両者を併用することができると考えられる．

7. まとめと今後の課題

本論文では，遅延評価における再帰的データ構造の特徴をとらえ，後続として現れる再帰的データのためのサンクを再利用する実装法を提案した．提案手法は，同じデータ構造が繰り返し現れることを利用し，再利用するサンクへの参照が単一であることを前提として，そのサンクが保持している値を更新して用いる．正格性解析などの既存の手法とほぼ直交する手法であるため，同時に用いることができる．提案手法の効果を確認するため，Grasgow Haskell Compiler に実装し，遅延評価を行うプログラムについて実験を行った．メモリ消費に関して十分な効果があることが分かった．

今後の課題は，以下の 3 項目である．

1 つ目の課題は，プログラム実行時間におけるオーバヘッドの削減である．まず，現状の実装では，4.2 節で説明したように，現状では再利用が可能であるかチェックしている．これらのチェックの中には，コンパイル時に省略することができる場合もあることが分かっている．次に，ごみ集めの実装に関して，さらなる検討が必要であると考えている．4.3 節で述べたように，再利用サンクは，世代別 GC の中で，特殊な扱いをしなくてはならない．現状では，再利用サンクを古い世代へと昇格させない実装を選んでいるが，別の方式などさらなる検討が必要である．

2 つ目の課題は，木などの複数箇所で再帰定義されるデータ構造に関して，本手法を適用することである．線形再帰的データ構造であれば，再利用できる可能性のあるサンクは一意に定まるため，リストと同様に適用可能である．しかし，二分木などの複数箇所で再帰的に定義されるデータ構造に関しては，どの式でサンクを再利用するかを決定しなくてはならない．たとえば，二分木であれば，左の子に対してサンクを再利用して用いると静的に決めてコンパイルすれば，提案手法をそのまま適用できる．実際には，どの再帰部分に対して再利用するのが効率的かは，コンパイル時に解析することが考えられる．

3 つ目の課題は，関連研究であげた Cheapness Analysis や Chunky Evaluation などの既存の手法と組み合わせた際の効果の測定を行うことである．現状の実験結果は，正格性解析と併用したものであったが，他の解析ともあわせて用いることができるはずである．

参考文献

[1] Hughes, J.: Why Functional Programming Matters, *The Computer Journal*, Vol.32, No.2, pp.98–107 (1989).

[2] Wadler, P. and Hughes, R.J.M.: Projections for Strictness Analysis, *Proc. Functional Programming Languages and Computer Architecture*, pp.385–407 (1987).

[3] Faxén, K.F.: Cheap eagerness: Specuative evaluation in a lazy functional language, *Proc. International Conference on Functional Programming*, pp.150–161 (2000).

[4] Bird, R.: *Introduction to Functional Programming using Haskell*, *2nd ed.*, Prentice Hall (1998).

[5] Peyton-Jones, S.: *The Implementation of Functional Programming Languages*, Prentice Hall (1987).

[6] Peyton-Jones, S.: Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine, *Journal of Functional Programming*, Vol.2, No.2, pp.127–202 (1992).

[7] Tolmach, A.: An External Representation for the GHC Core Language, available from ⟨http://www.haskell.org/ghc/docs/papers/core.ps.gz⟩ (accessed 2010-10-01) (2001).

[8] Peyton-Jones, S., Hall, C., Hammond, K., Partain, W. and Wadler, P.: The Glasgow Haskell Compiler: A Technical Overview, *Proc. Joint Framework for Information Technology Technical Conference*, pp.249–257 (1993).

[9] Peyton-Jones, S. and Partain, W.: Let-floating: Moving Bindings to Give Faster Programs, *Proc. International Conference on Functional Programming*, pp.1–12 (1996).

[10] Partain, W.: The `nofib` Benchmark Suite of Haskell Programs, *Proc. Glasgow Workshop on Functional Programming*, pp.195–202 (1992).

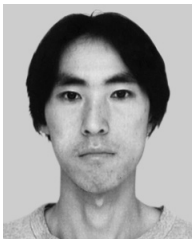
[11] Plasmeijer, R. and van Eekelen, M.: Concurrent Clean Language Report – Version 1.3, Technical Report CSI-R9816, University of Nijmegen (1998).

[12] Barendsen, E. and Smetsers, S.: Conventional and Uniqueness Typing in Graph Rewrite Systems, *Proc. IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, pp.41–51 (1993).

[13] Hage, J., Holdermans, S. and Middelkoop, A.: A Generic Usage Analysis with Subeffect Qualifiers, *Proc. International Conference on Functional Programming*, pp.235–246 (2007).

[14] Launchbury, J., Gill, A., Hughes, J., Marlow, S., Peyton-Jones, S. and Wadler, P.: Avoiding Unnecessary Updates, *Proc. Glasgow Workshop on Functional Programming*, pp.144–153 (1992).

[15] Ennals, R. and Peyton-Jones, S.: Optimistic Evaluation: An Adaptive Evaluation Strategy for Non-Strict Programs, *Proc. International Conference on Functional Programming*, pp.287–298 (2003).



高野 保真　（正会員）

1981 年生．2004 年電気通信大学電気通信学科情報工学科卒業．2006 年電気通信大学大学院電気通信学研究科情報工学専攻修士課程修了．2008 年株式会社コマ・システムズを設立．修士（工学）．日本ソフトウェア科学会，ACM 各会員．

岩崎 英哉　（正会員）

1983 年東京大学工学部計数工学科卒業．1988 年東京大学大学院工学系研究科情報工学専攻博士課程修了．同年同大学計数工学科助手．1993 年同大学教育用計算機センター助教授．その後，東京農工大学工学部電子情報工学科助教授，東京大学大学院工学系研究科情報工学専攻助教授，電気通信大学情報工学科助教授を経て，2004 年より電気通信大学教授．工学博士．プログラミング言語，システムソフトウェア等の研究に従事．日本ソフトウェア科学会，ACM 各会員．

鵜川 始陽　（正会員）

1978 年生．2000 年京都大学工学部情報学科卒業．2002 年京都大学大学院情報学研究科通信情報システム専攻修士課程修了．2005 年京都大学大学院専攻博士後期課程修了．2005 年京都大学大学院情報学研究科特任助手．2008 年より電気通信大学助教．博士（情報学）．プログラミング言語とその処理系に関する研究に従事．