情報処理学会論文誌　プログラミング　Vol. 1　No. 2　28–41 (Sep. 2008)

純関数型言語の処理系における効率的な枝刈り機構の実装

田村知博[†1]　高野保真[†1]　岩崎英哉[†2]

実行効率の良い探索プログラムを書くためには，結果に寄与しない計算を除去する枝刈りが有効であるが，一般に，簡潔なプログラム構造を保ったまま枝刈りの記述をすることは難しい．この問題点を解決するために，Improving Sequence（IS）というデータ構造を用いる手法が提案され，有効性が確認されている．IS とは，ある全的に定義された推移的な二項関係に従い，要求駆動によって単調に最終結果へ近づいていく近似値の列のことである．IS を用いたプログラムでは，枝刈りをするか否かの判断に近似値を用いることができる．従来の純関数型言語上の IS の実装はライブラリによるものだったので，近似値の数に比例したヒープ領域を必要とし，効率があまり良くないという問題点があった．この問題点を解決するため，本論文では，IS の近似値が持つ単調性に着目し，また，近似値が枝刈りの判断にのみ用いられ，中間データとしての役割しか持たないことを前提として，純関数型言語において，定数領域しかヒープを消費しない IS の実装を提案する．提案手法の効果を確認するため，Glasgow Haskell Compiler に実装を施し，ナップサック問題など，いくつかのプログラムで実験を行った．その結果，問題によってばらつきはあるものの，メモリ消費量において 3%〜75%程度の減少が見られ，安定的に改善できることが確認できた．また，メモリ消費量を削減することで，多くの場合に実行時間を改善できることも確認できた．

An Efficient Implementation of Pruning Mechanism in a Purely Functional Programming Language

Tomohiro Tamura,[†1] Yasunao Takano[†1]
and Hideya Iwasaki[†2]

Pruning unnecessary computations increases the efficiency of search programs. However, in general, it is hard to write a program that performs pruning while retaining program clarity. A mechanism called Improving Sequence (IS for short) helps the programmers write clear program for pruning computations. An IS is a sequence of approximations that are gradually improved based on a binary relation and approach the final value. Each approximation value can be used to decide whether further computations beyond the approximation can be pruned or not. Existing implementation of an IS on a purely functional language is library-based one, thus much heap space that is proportional to the number of approximations is consumed. To solve this problem, this paper proposes an efficient implementation of an IS that consumes only the constant space within the heap space. The proposed implementation is based on the observation that the approximations with an IS is monotonic and each approximation is used only in the judgment of pruning. We implemented the proposed idea in the Glasgow Haskell Compiler, and made some experiments by running some search programs that use ISes. The results show that the amount of memory consumption is reduced by 3–75%, depending on the program, and the execution time is also reduced in many cases.

1. はじめに

実行効率の良いプログラムを書くためには，結果に寄与しない計算を除去する枝刈りが有効である．枝刈りを記述するためには，問題の性質に合わせて，計算を除去してもよい場合の条件を明示する必要がある．そのため，一般に，簡潔なプログラム構造を保ったまま枝刈りの記述をすることは難しい．この問題点を解決するために，Improving Sequence（以下，IS と呼ぶ）というデータ構造を用いる手法[1)] が提案され，現在，IS を用いた枝刈りの有効性が確認されている[4)]．

IS とは，全的に定義された推移的なある二項関係に従い単調に最終結果へ近づいていく近似値の列のことである．IS の簡単な説明は 2 章で述べる．IS を実現するためには，プログラム言語が遅延評価機構を備えている必要がある．IS を実現した既存の実装には，Scheme 処理系[7)] を拡張し遅延評価機構を組み込んだもの[6)] と，もともと遅延評価機構を持つ言語 Haskell[5)] 上のライブラリとして実現したもの[4)] があった．

しかし，既存の実装には次のような問題点がある．Scheme 処理系を拡張したものは，拡張する元とした処理系が一般に広く使われているものではない点に問題がある．また，Haskell 上のライブラリとして実装されたものでは，ライブラリであるがゆえに，IS のメモリ効率

†1 電気通信大学大学院電気通信学研究科情報工学専攻
Department of Computer Science, Graduate School of Electro-Communications, The University of Electro-Communications
†2 電気通信大学情報工学科
Department of Computer Science, The University of Electro-Communications

に問題がある．

そこで本研究では，一般的な遅延評価機構の上で働く枝刈り機構の効率的な実装を目的とし，その実現のために，Haskell 上に効率の良い IS を実現する．具体的には，代表的な Haskell コンパイラである Glasgow Haskell Compiler（GHC）を拡張し，IS を処理系組み込みのデータ構造として，IS に対してメモリ効率の良いコードを生成する．このことにより，IS を用いたプログラムのメモリ消費量が削減される．また，ごみ集めの処理が軽減されることで実行時間の減少を見込むことができる．

以下，本論文の構成を述べる．2 章では，IS の定義や枝刈りの仕組みなどについて簡単に述べる．3 章では GHC について述べる．また，4 章で提案機構の設計を，5 章で提案機構の実装について述べ，6 章では提案機構の評価のために行った実験について述べる．7 章で本論文をまとめる．

2. Improving Sequence

本章では，IS の定義・性質について述べ，IS を利用した枝刈りについて説明する．また，IS の既存実装について特徴を述べ，問題点を明らかにする．

2.1 IS による枝刈り

IS とは，近似値と最終結果からなる列のことである．隣り合う近似値および最終結果の間には，全的に定義された推移的なある二項関係が成り立ち，近似値はその二項関係に従って単調に最終結果へ近づいていく．

IS は次の記法を用いて記述する．

$$a_0 \ll_R a_1 \ll_R a_2 \ll_R \cdots \ll_R a_n \ll_R \varepsilon \tag{1}$$

IS は，IS 構成子 $\ll_R$ と IS の終端記号 ε を用いて記す．ここで，R は a_i と a_{i+1}（$0 \leq i \leq n-1$）の間に成り立つ二項関係を表す．(1) の IS は，近似値 a_0 から始まり，a_1，$a_2 \cdots$ と，二項関係 R に従って単調に最終結果 a_n へと近づく IS である．簡単のため本論文では以降，二項関係 R を比較演算子 $<$ に限定し，省略して $\ll$ と表記する．

IS の計算は要求駆動によって進められる．よって，近似値は 1 度にすべて計算されるわけではなく，必要に応じて 1 つずつ計算される．(1) の IS において，a_0 のみが計算された段階では，a_1 以降の近似値は計算されずに（計算が遅延されて）残っている．この状態を以下のように表記する．

$$a_0 \ll rest \tag{2}$$

rest の部分には，後続計算を表す式が入るものとする．ここで，a_0 の次の近似値が必要になれば *rest* の部分が要求駆動的に計算される．その結果，以下のように次の近似値 a_1 が得られ，さらにその後続計算は遅延される．

$$a_0 \ll a_1 \ll rest' \tag{3}$$

ここで $rest'$ は a_1 に続く遅延された後続計算である．すでに計算された最新の近似値は，枝刈りの判断に用いることが可能である．たとえば (2) においては a_0 を，(3) においては a_1 を枝刈りの判断に用いることができる．

次に，IS を用いた枝刈りについて，例をあげて説明する．**図 1** (a) に，リストの長さを求める関数 *length* の，IS を用いない定義を示す．*length* は 2 つの引数をとり，第 1 引数は長さを求めたいリスト，第 2 引数はそれまでに求めたリストの長さを保持する累積変数である．図 1 の 1 行目は，リストが空である場合，その時点での累積変数 n が最終結果であることを示している．また，2 行目は，リストが空でない場合，リストの先頭の要素を取り除いたものと，累積変数 n を 1 増加したものを *length* に与えるという再帰的な定義を示している．

$$\begin{aligned} &length\ [\,] \quad\ \ n &&= n \\ &length\ (x:xs)\ n &&= length\ xs\ (n+1) \end{aligned}$$

(a) IS を構成しない通常の *length* の定義

$$\begin{aligned} &length\ [\,] \quad\ \ n &&= n \ll \varepsilon \\ &length\ (x:xs)\ n &&= \underline{n \ll} \ length\ xs\ (n+1) \end{aligned}$$

(b) IS を構成する *length* の定義

図 1　*length* の定義
Fig. 1　Definition of *length*.

続いて，IS を構成するようにした *length* の定義を図 1 (b) に示す．ここで，図 1 (a) との違いに下線を引いて示している．1 行目には，最終結果であることを表すための「$\ll \varepsilon$」が加わっている．また，2 行目には，近似値を示すための「$n \ll$」が加わっている．2 行目において，$length\ xs\ (n+1)$ の部分が後続計算にあたる．

図 1 のように，元の定義の簡潔な構造を保ったまま，IS を構成する定義へと容易に変更することができる．

ここで，IS を構成するよう定義された図 1 (b) の *length* において，どのように枝刈りが行われるかを例を用いて示す．

まず，通常の *length* の計算がどのように進むかを示すため，1 から 100 までの整数からなるリスト [1..100] の長さと，整数 1 との比較を考える．計算の様子を**図 2** (a) に示す．図 2 (a)

$$
\begin{array}{ll}
\quad 1 \ < \ \mathit{length}\ [1..100]\ 0 & \quad 1 \ <_{IS} \ \mathit{length}\ [1..100]\ 0 \\
\Rightarrow 1 \ < \ \mathit{length}\ [2..100]\ 1 & \Rightarrow \underline{1 \ <_{IS} \ 0} \ \ll \ \mathit{length}\ [2..100]\ 1 \\
\Rightarrow 1 \ < \ \mathit{length}\ [3..100]\ 2 & \Rightarrow \underline{1 \ <_{IS}} \ 0 \ \ll \ \underline{1} \ll \ \mathit{length}\ [3..100]\ 2 \\
\Rightarrow 1 \ < \ \mathit{length}\ [4..100]\ 3 & \Rightarrow \underline{1 \ <_{IS}} \ 0 \ \ll \ 1 \ll \ \underline{2} \ \ll \ \mathit{length}\ [4..100]\ 3 \\
\quad \cdots & \Rightarrow \mathit{True} \\
\Rightarrow 1 \ < \ 100 & \\
\Rightarrow \mathit{True} &
\end{array}
$$

(a) 通常の *length* の評価過程　(b) IS を構成する *length* の評価過程

図 2　*length* の評価過程
Fig. 2　Evaluation of *length*.

では，関数 *length* を再帰的に 100 回適用し，リストの長さが整数 100 であることを得てから，1 との比較を行い，最終結果 *True* を得ている．

次に，IS を構成するよう定義された *length* を用いた計算過程を図 2 (b) に示す．図 2 (a) と同様に，$1 <_{IS}$ *length* [1..100] 0 という式を考える．ここで比較演算子を $<_{IS}$ としたのは，第 2 オペランドが IS であり，通常の比較演算子 $<$ と異なることを明示するためである．

通常の *length* と異なり，IS を構成する *length* では，各近似値と整数 1 との比較が可能である．1 行目の *length* [1..100] 0 という式を計算すると，$0 \ll$ *length* [2..100] 1 という IS が構成される．このとき，$<_{IS}$ によって近似値 0 を取り出し 1 と比較する．この比較の結果は偽であるが，近似値 0 が最終結果かどうか分からないため，後続計算である *length* [2..100] 1 の計算結果は 1 より大きくなる可能性がある．そのため計算を進める要求が生じ，遅延されていた *length* [2..100] 1 が計算される．*length* [2..100] 1 は $1 \ll$ *length* [3..100] 2 という IS を構成する．次の近似値 1 が求まったため，$<_{IS}$ の第 1 オペランドである 1 との比較がなされるが，その結果は偽であるため，先と同様にこの時点でも判断をつけることはできない．そのため後続する *length* [3..100] 2 が計算される．*length* [3..100] 2 は $2 \ll$ *length* [4..100] 3 という IS を構成し，1 と近似値 2 の比較がなされる．ここで，1 は近似値 2 よりも小さく，また，IS の後続する近似値は（存在するならば）$<$ に基づいて単調に大きくなっていくことから，近似値 2 を計算した段階で，*length* [1..100] 0 の最終結果は少なくとも 2 以上であることが分かる．そのため，1 と近似値 2 との比較を行った時点で，2 以降の後続計算を行うことなく，最終結果 *True* を返すことができる．その結果，*length* [4..100] 3 という後続計算が枝刈りされる．

length の例では，枝刈りを行うのは数値と IS をオペランドとする比較演算子 $<_{IS}$ である．$<_{IS}$ は，IS を少しずつ読み進め，さらに IS を読み進めるかどうかの判断を行う．このような比較演算子 $<_{IS}$ の定義を**図 3** に示す．

$$
\begin{array}{l}
n \ <_{IS} \ \varepsilon \ = \ \mathit{False} \\
n \ <_{IS} \ (x \ \ll \ xs) \ = \ \mathit{if}\ n < x\ \mathit{then}\ \mathit{True}\ \mathit{else}\ n \ <_{IS} \ xs
\end{array}
$$

図 3　枝刈りを行う比較演算子 $<_{IS}$
Fig. 3　Comparison operator $<_{IS}$ for pruning.

$$
\begin{array}{l}
\mathit{minB}\ (x \ll xs)\ \varepsilon \ = \ \varepsilon \\
\mathit{minB}\ \varepsilon\ (y \ll ys) \ = \ \varepsilon \\
\mathit{minB}\ (x \ll xs)\ (y \ll ys) \ = \ \mathit{if}\ (x < y)\ \mathit{then}\ x \ll \mathit{minB}\ xs\ (y \ll ys) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else\ if}\ (x > y)\ \mathit{then}\ y \ll \mathit{minB}\ (x \ll xs)\ ys \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{else}\ x \ll \mathit{minB}\ xs\ ys
\end{array}
$$

図 4　枝刈りを行う関数 $minB$
Fig. 4　Function $minB$ for pruning.

この例で示したように IS の利用においては，IS を構成するよう変更した関数と，$<_{IS}$ のような枝刈りを行う基本演算とを組み合わせて枝刈りを実現することができる．

IS を利用した枝刈り機構では，問題の定義である IS を構成するプログラムと枝刈りの記述が分離できているため，枝刈りを行う基本演算の再利用が可能である．**図 4** に，基本演算の 1 つである関数 $minB$ の定義を示す．$minB$ は，引数に 2 つの IS をとり，両者の最終結果の小さい方を最終結果とするような IS を構成して返す．そのために $minB$ は，両者の近似値を比較してマージし，値の小さい方を優先的に読み進める．また，どちらかの IS が終端に達したなら，戻り値の IS も終端となる．つまり，近似値においては，より小さいものが先に採用され結果の IS として構成されていき，最終結果に関しても，小さいものが選ばれることになる．そのため，探索問題において探索空間を IS で表現すれば，最小値を求めるような場面で IS と $minB$ を用いることで，枝刈りが実現できる．

2.2　IS の既存の実装

本節では，IS の既存の実装について述べ，それぞれの問題点を明らかにする．

2.2.1　Scheme 処理系を拡張した実装

Scheme 処理系を拡張した実装[6] は，Java により実装された Scheme インタプリタ[7] を基にして拡張を施した，Scheme から Java へのコンパイラである．この処理系では，IS は処理系組み込みのデータ構造として実装されており，IS のデータ構成子 $\ll$ に対応する特殊形式 `$<<` を用い，以下のように記述して構成する．

`($<<` 二項関係 近似値 後続計算`)`

Scheme は先行評価を基本とするため，通常の関数であれば引数をすべて評価してから関数適用を行うが，`$<<` は後続計算にあたる部分の式を遅延させる（二項関係と近似値の部分

は遅延されない）特殊形式である．遅延された式は，値が必要になったときに計算される．この特殊形式により，IS の計算を要求駆動によって進めることが可能になっている．

この実装では，拡張する元となった Scheme 処理系が，一般に広く使われているものではない点に問題がある．さらに，本来は先行評価である Scheme に，IS の実現のために遅延評価を導入した点にも問題がある．

2.2.2　Haskell 上のライブラリとしての実装

この実装[4] は，純関数型言語 Haskell 上で，通常の（組み込みでない）データ構造として IS を実現している．Haskell は遅延評価が標準であるため，処理系の拡張は行われていない．

このライブラリでは，IS はリストに似たデータ構造として次のように定義されている．

```
data Ord a => IS a = E | a :? (IS a)
```

ここで `E` は ε に，`:?`は $\ll$ に対応するデータ構成子である．また，隣接する 2 つの近似値が満たすべき二項関係は，`Ord` クラスにおける `<` を用いる．後続計算，つまりリストでいえば後尾の要素にあたる部分の式はデフォルトで遅延されるため，IS に必要な要求駆動が自然に実現されている．

このようにライブラリとして実装された IS は，ライブラリであるがゆえに IS によって消費されるメモリに無駄がある．GHC において IS がどのようにメモリを消費するかについての詳細は 3 章で説明するが，要点を以下に述べる．

IS の近似値には単調性があるため，ある近似値 a_i を考えたとき，次の近似値 a_{i+1} は IS の二項関係の意味において a_i よりも最終結果に近い．そのため，枝刈りの判断をする目的で近似値を用いる際に，a_i の代わりに a_{i+1} を用いれば，より良い判断が行える．よって，a_{i+1} が計算された時点において，古い近似値 a_i は不要になっている．ところが Haskell 上のライブラリとしての実装では，a_i と a_{i+1} のそれぞれにメモリ領域を割り当てるため，効率が悪くなっている．

3.　Glasgow Haskell Compiler

本研究では，前節で述べた問題点を解決するため，IS を Haskell 処理系組み込みのデータ型として実装する．このことにより，Haskell 上で動作する，それぞれの近似値に対して新たにメモリを消費しない IS を実現した．実装は Haskell の標準的なコンパイラである Glasgow Haskell Compiler（GHC）上に行った．

GHC は遅延評価機構を持つ言語処理系として標準的なものであり，この処理系上で効率的な IS を実現することは重要である．処理系を拡張するため，ライブラリによる実現のような汎用性は失われるが，本研究では効率的な IS の実装が可能であることの確認，およびその効果の測定を研究の主眼とした．

本章では，GHC（バージョン 6.6.1）におけるデータ構造の内部表現，評価過程，およびコンパイルの概要について述べる．また，GHC で採用されているごみ集め機構についても説明する．

3.1　サンクやデータ構造の内部表現

遅延された計算を表現する処理系内部のデータ構造をサンク（thunk）という．GHC では，サンクや，Haskell 上で定義されたデータ構造のように，ヒープ領域へ動的に割り当てられるデータ（以降，ヒープオブジェクトと呼ぶ），および後述するスタックフレームに対して，すべて同一の構造を持たせている．そのため，GHC ではこれらを総称してクロージャと呼ぶ．**図 5** にクロージャの構造を示す．

クロージャの先頭 1 ワードは info pointer と呼ばれ，info table という表へのポインタが収められている．info table には，関数定義へのポインタ（コードポインタと呼ぶ），クロージャタイプ（サンク，スタックフレームなどを区別するためのタグ），保持しているポインタ数が収められている．クロージャの残りの領域は payload と呼ばれ，サンクであれば計算に必要な他のヒープオブジェクトへのポインタが収められており，リストのようなデータ構造であれば保持するヒープオブジェクトへのポインタが収められている．

例として，図 1 (a) の *length xs* $(n+1)$ に対応するサンクの構造を**図 6** に示す．このサンクに対応する info table には，*length* に対応するコードポインタや，クロージャがサンクであることを示す `THUNK` というタグ[⋆1]，ポインタ数 2 が収められている．ここで，破線で

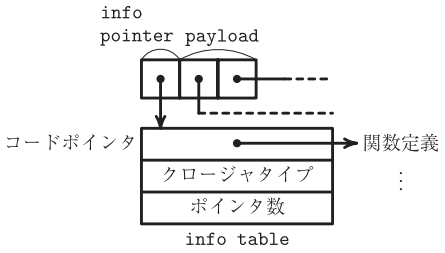


図 5　クロージャの構造
Fig. 5　Structure of closure.

⋆1 GHC のソースコード中で，`#define THUNK 16` のように定義されている．

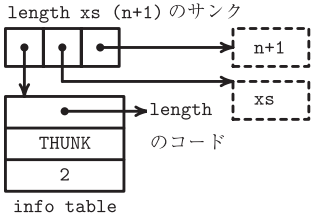
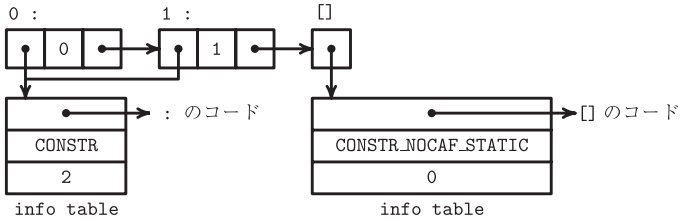
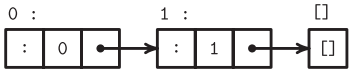


図 6　length xs (n+1) のサンク
Fig. 6　Thunk: length xs (n+1).



(a) 0:1:[] に対応するデータ構造



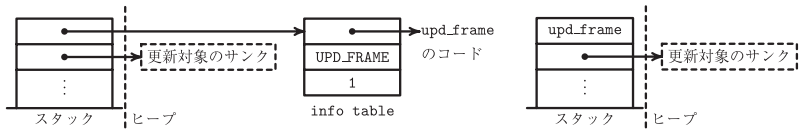
(b) 省略表記

図 7　リストの構造
Fig. 7　Structure of 0:1:[].

描かれたオブジェクトは，式に対応するサンクを意味している．

もう 1 つの例として，0:1:[] に対応するリストのデータ構造を**図 7** (a) に示す．: は Cons を意味する中置データ構成子であり，[] は Nil を意味する．図 6 に示した構造とほぼ同じであり，: に対応する info table には : に対応するコードポインタと，データ構成子であることを示す CONSTR というタグ，ポインタ数 2 が収められている．また，[] に対応する info table には，静的に割り当てられるクロージャであることを示す CONSTR_NOCAF_STATIC のタグと，ポインタの数 0 が収められている．[] のようなデータ構造は動的に割り当てる必要がないため，[] に対応するクロージャはすべてのリストで共有される．

以降，本論文では，図の簡略化のために，info table を info pointer の中に略記する．その例を図 7 (b) に示す．info pointer の部分には，対応する関数名・データ構成子名を書いてこれに代えるものとする．



(a) 更新フレーム　(b) 省略表記

図 8　スタックフレーム
Fig. 8　Stack frame.

3.1.1　スタックフレーム

GHC では，ソースコードを最終的に C 言語へ変換する．しかしオブジェクトを積むスタックの管理を C 言語の処理系に任せず，GHC によって管理するコードを出力する．これは主にごみ集め機構のためであり，スタックにおけるポインタの所在を明らかにするためである．

スタックフレームは，ヒープオブジェクトと同様の構造を持っており，スタックフレームの先頭が，ヒープオブジェクトの先頭 1 ワード（すなわち info pointer）に相当する．

例として，サンクの更新を行うスタックフレーム（upd_frame）の構造を**図 8** (a) に示す．upd_frame は，サンクの再計算を防ぐ目的で，サンクが評価された場合に評価結果への間接ポインタで上書きするために積まれるフレームである．

スタックトップには info table へのポインタが，次の 1 ワードには更新対象のサンクへのポインタが収められている．また，info table には，upd_frame の定義（すなわち，サンクの更新を行うコード）へのポインタや，更新を行うフレームを示す UPD_FRAME のタグ，ポインタ数 1 が収められている．

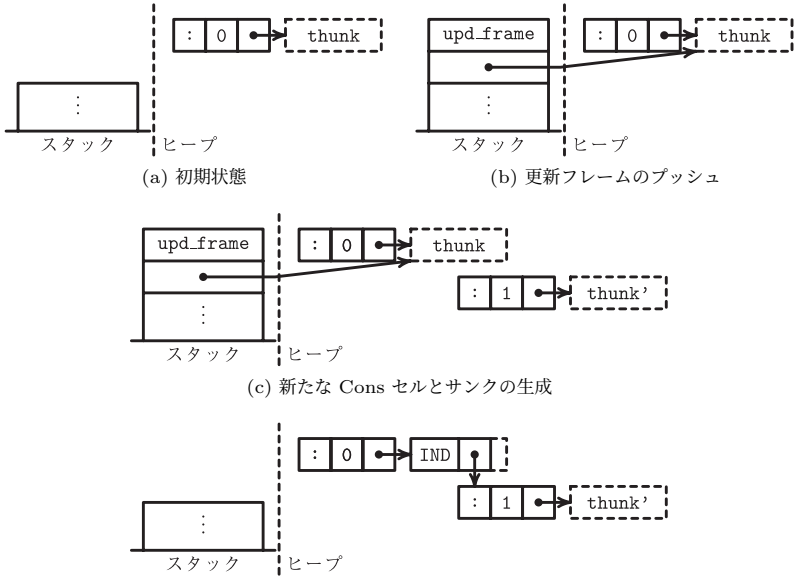
ヒープオブジェクトと同様に，本論文では以降スタックフレームに対しても省略して表記する．その例を図 8 (b) に示す．

3.2　評価過程

ここで，リストを例にとり，サンクおよびデータ構造の割当てがどのように起こるかを説明する．

図 9 (a) は，0:thunk という Cons セルがすでに割り当てられた状態である．ここで，thunk が評価されると 1:thunk' が生成されるものとする．

thunk の値が要求されると，まずスタックに更新フレーム（upd_frame）が積まれ，thunk を更新する準備がなされる（図 9 (b)）．そののちに，thunk の評価を行う．すると，ヒープ



(a) 初期状態　(b) 更新フレームのプッシュ

(c) 新たな Cons セルとサンクの生成

(d) 古いサンクの上書き，更新フレームのポップ

図 9　リストの計算過程
Fig. 9　Evaluation of linked list.

に新しく thunk' と Cons セル（1:thunk'）が割り当てられる（図 9 (c)）．ここで割り当てられた Cons セルが thunk の評価結果として返され，スタックトップにある upd_frame が実行される．upd_frame では，更新対象として積んであったポインタをたどり，対象の thunk を評価結果への間接ポインタで上書きする．図では，IND によって間接ポインタであることを示している．最後に，割り当てられた Cons セルを評価結果として返し，upd_frame をスタックから取り除く（図 9 (d)）．

図中の IND は，間接ポインタのために用意された関数であり，IND クロージャの値に対して要求が出された場合には，クロージャの持つ間接ポインタをそのまま返すという定義になっている．GHC では，ごみ集めの際に IND クロージャを取り除き，間接ポインタをなくす処理を行う．そのため，IND クロージャは，生成された直後のごみ集めで回収される．GHC におけるごみ集めについては，次節で詳しく述べる．

Haskell 上のライブラリとして実装された IS は，リストと同様の形で実現されているため，リストと同様に，後続計算のサンクを評価することで新たな IS やサンクの割当てが起こる．

3.3　ごみ集め

GHC 6.6.1 では，ごみ集めアルゴリズムに世代別コピー方式を採用している．世代別ごみ集めでは，生存データを殿堂入りさせる時期と，世代数を決める必要があるが，GHC のデフォルトでは，世代数は新旧の 2 世代であり，データが生成されてから 2 回目にごみ集めがされたときに，そのデータは新世代から旧世代へと移される．

世代別ごみ集めで留意すべき点は，旧世代から新世代へのポインタ（逆方向ポインタ）を管理する必要があるということである．これは，新世代のみを対象としたごみ集めの際には旧世代領域のポインタを追跡しないので，逆方向ポインタを管理しておかないとデータの必要性が正しく判断されないためである．

一般に，逆方向ポインタは旧世代にあるデータを破壊的に更新したとき発生する．Haskell では，再計算を防ぐために行われるサンクの更新によって逆方向ポインタが発生しうるため，逆方向ポインタを適切に管理する必要がある．

逆方向ポインタの管理には，リメンバセットというデータ構造を用いている．これは，旧世代から新世代へのポインタが発生したときに，それを保持しておくためのデータ構造である．GHC では，実行時システムで定義された mut_list というデータ構造がリメンバセットにあたる．mut_list は，最も若い世代を除いて世代それぞれに存在し，ごみ集めの対象となった世代の mut_list は破棄される．

もう 1 つごみ集めで留意すべき点に，サンクを原因とするスペースリークの問題がある．以下，この問題と，GHC における解法について説明する．

あるサンクの値が要求されると，スタックに更新フレームが積まれ，後の間接ポインタへの更新のためにサンクへのポインタが保持される．更新を行うため，サンクの占める領域自体は必要だが，サンクが内部に保持しているポインタが指す領域は，すでに不要となっていることがある．しかし，サンクがスタック上から指されているために，サンクからたどれるこれらの不要な領域は必要と判断され，再利用されずに残る．これがサンクを原因とするスペースリークの問題である．

GHC では，Blackholing [2] という手法でこれを回避している．Blackholing とは，更新フレームから指されているサンクに印をつけ，ごみ集めのときにたどらないようにする，という手法である．GHC では，ごみ集めを始める前にスタックを走査し，更新フレームから指されているサンクの info pointer を書き換えることで Blackholing を実現している．す

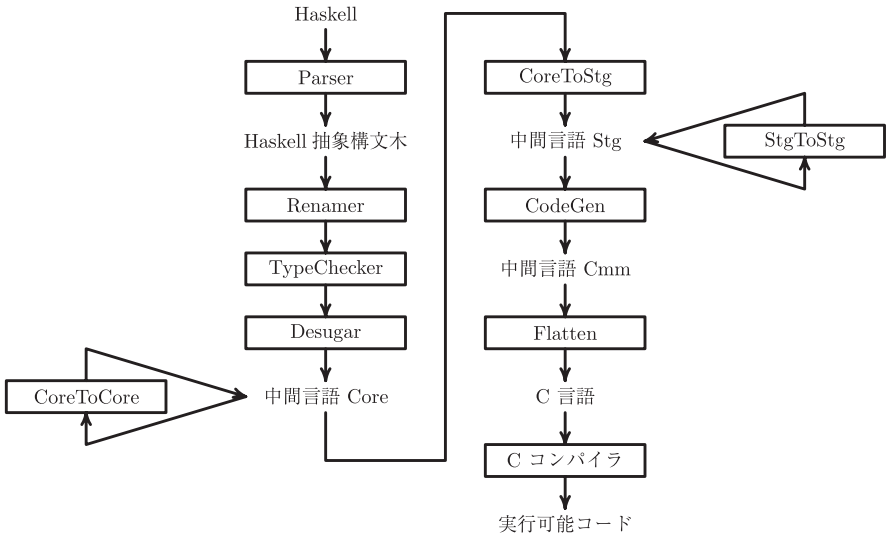


図 10　コンパイルの概要
Fig. 10　Compile overview.

なわち，更新フレームから指されているサンクの info pointer は，Blackholing がなされていることを示す info table を指すよう変更される．この Blackholing 用 info table の情報を元にしてごみ集めが行われるため，ごみ集めの際にスタックから指されているサンクは，ポインタの追跡がなされない．

3.4　コンパイルの概要

本節では，GHC におけるコンパイルの概要を簡単に述べる．まず，大まかな流れを**図 10** に示す．

Haskell のソースコードは，まず Parser によって抽象構文木となる．次に，Renamer によってスコープの解決が行われ，必要に応じて名前のつけかえがなされる．その後，TypeChecker が型の推論・検査をする．型の不整合があれば，この時点でエラーとなる．型チェックが終わると，Desugar が Haskell の構文糖衣を取り去り，中間言語 Core に変換する．以降は，中間言語 Stg，中間言語 Cmm と変換し，最終的には C 言語のコードを出力する．これを C コンパイラ（GCC）でコンパイルし，GHC によってあらかじめ用意された実行時システムと結合して，実行可能コードとなる．中間言語 Core および中間言語 Stg の段階では，最適化のための変換がいくつか行われる．

4. 設　　計

2.2.2 項で述べたように，IS の近似値には単調性があるため，近似値が枝刈りの判断にのみ用いられ，中間データとしての役割しか持たないという前提のもとで，IS に割り当てられたメモリ領域をつねに上書きして古い近似値を捨ててもかまわない．

新たな IS の割当ては，IS の後続計算にあたるサンクを評価することで行われる．そのため，このサンクの評価過程に手を加え，新たに IS を生成しないようにすることを考える．従来の評価過程では，新たに IS と後続計算のサンクを生成した後で，間接ポインタで元のサンクを上書きしていたが，提案機構では上書きの方法を変更し，新たに IS およびサンクを生成することなく，すでに割り当てられた領域を上書きする．つまり，IS は新しい近似値を指すように，サンクは新たな後続計算を保持するように書き換える．

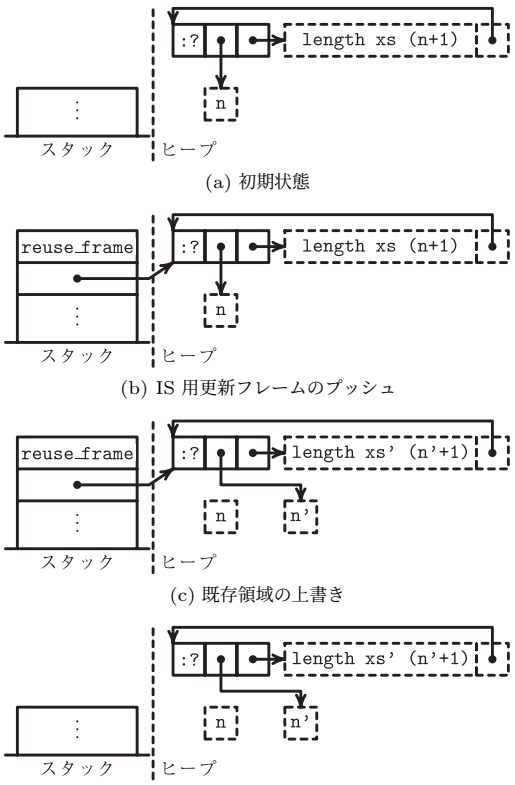
ここで留意すべき点は，IS は固定長であるため IS の領域はつねに上書きできるが，サンクは可変長オブジェクトである（後続計算の形によって大きさが変わる）ため，サンクの領域は上書きできない場合があることである．サンクの上書きができる例としては，図 1 (b) の関数 *length* があげられる．*length* の場合には，後続計算がつねに *length xs* $(n+1)$ という形をしているため，サンクの構造にも変化がなく，つねにサンクの領域を再利用できる．サンクの上書きができない例を次に示す．

$$IS \;=\; 0 \;\ll\; f\;1$$
$$f\;x \;=\; x \;\ll\; g\;x\;2$$

この IS において，近似値 0 が計算された段階での後続計算は $f\;1$ という式である．$f\;1$ のサンクは，f に対応する info pointer と数値 1 を保持するための 2 つの領域を持つ．ところが，$f\;1$ を評価した結果に現れる新たな後続計算は $g\;x\;2$ という式であり，この式のサンクには g に対応する info pointer，引数 x，数値 2 の 3 つの領域が必要となる．これは $f\;1$ のサンクの領域よりも大きいため，$f\;1$ のサンクを上書きして再利用することができない．このような場合には，サンクの上書きは諦めて，サンクのみ従来どおり新たに生成し，IS の上書きだけを行う．

以下，提案機構での IS およびサンクの再利用方法を具体的に述べる．

本機構を可能とするため，あらかじめ後続計算のサンクに，更新対象である IS へのポインタを追加しておく（**図 11** (a)）．これは，IS とサンクは独立したヒープオブジェクトであることから，従来のサンクの持つ情報からは上書きして再利用すべき IS の位置が分からな



(a) 初期状態

(b) IS 用更新フレームのプッシュ

(c) 既存領域の上書き

(d) IS 用更新フレームのポップ

図 11　提案機構による IS の計算過程
Fig. 11　Evaluation of IS with proposed mechanism.

いためである．図 11 (a) は $n \ll length\ xs\ (n+1)$ という IS が，すでにヒープ中に割り当てられている状態を示している．図中 `:?` は $\ll$ に対応する info table を表す．こうして変更したサンクを，従来と同じ方法で評価を始める．すなわち，info table のコードポインタが指すコードを実行する．

次に，サンクが評価されるときの，スタックに積むフレームを変更する必要がある（図 11 (b)）．従来の更新処理とは異なる処理が必要になるため，間接ポインタを作成する従来の更新フレーム（`upd_frame`）ではなく，本機構で新たに用意した IS 用更新フレーム（`reuse_frame`）を積む．また，従来はサンクへのポインタをスタックに保持していたが，提案機構ではサンクに追加した IS へのポインタを積むようにする．ここで，スタックフレームを積む処理は，サンクに対応するコード（すなわち，図 5 でコードポインタが指しているコード）に含まれている．そのため，IS 用更新フレームを積むためには，サンクのコードの該当部分を変更すればよい．

サンクのコードには，新しい近似値のための IS および後続計算のサンクの生成処理も含まれる．そのため，この部分に変更を加え，まずスタックを見て，積まれているのが従来の更新フレーム（`upd_frame`）か IS 用更新フレーム（`reuse_frame`）かで分岐を行うようにする．もし積まれているのが `upd_frame` であれば，従来どおりの処理を行い，IS とサンクを生成する．ただし，サンクには生成する IS へのポインタを持たせる．積まれているのが `reuse_frame` であれば，スタックから IS へのポインタを取り出し，既存の IS 領域を上書きする．また，ここで古いサンクの領域が新しいサンクの内容で上書きするのに十分な大きさであれば，サンクの領域に対しても上書きを行い，再利用する．図 11 (c) が，サンクの上書きを行った場合となる．そして，上書きした IS を，評価結果として返す．

上記のように提案機構では，サンクのコードを変更することと，IS 用更新フレームを追加することによって，新たにメモリを消費しない IS を実現する．変更したサンクの擬似コードおよび IS 用更新フレームの擬似コードは，次章に掲載して説明する．

5. 実　　装

本章では，提案機構を実現するために GHC 6.6.1 へ加えた変更点について述べる．

5.1　コンパイラ部の変更

提案機構の実現には，まず GHC のコンパイラ部を変更する必要がある．上書きの可能な部位を特定し，特定した部位に対して前章で述べたようなコードを生成しなければならない．

上書き可能な部位の特定は，中間言語 Stg から Stg への変換によって行うようにした．中間言語 Stg の段階では，関数適用およびデータ構成子適用の引数には変数もしくは定数しかない形になっている．Haskell のソースコード上で適用の引数に与えられていた式は，中間言語 Core の最後の段階でくくり出され，let 式によって前もって束縛が与えられるようになる．

たとえば，図 1 (b) の 2 行目右辺に対応する，**図 12** (a) のような Haskell のコードを考える．このコードは，中間言語 Stg の段階では図 12 (b) のように変換されている．Haskell のコード上でデータ構成子 `:?` に与えられていた `length xs (n+1)` という式が前方にくく

```
n :? length xs (n+1)
```

(a) Haskell

```
StgLet len_thunk = length xs (n+1)
in StgConApp (:?) n len_thunk
```

(b) Stg

```
StgReusableLet len_thunk = length xs (n+1)
in StgReusableConApp (:?) n len_thunk
```

(c) 提案機構によって変換した Stg

図 12　中間言語 Stg
Fig. 12　Intermediate language: Stg.

り出され，let 式（StgLet）によって束縛を与えられている．また，中置データ構成子適用が前置記法（StgConApp）に変換されている．

このように，中間言語 Stg の段階では，サンクに対して let 式で束縛が明示された形になっている．また，この let 式は，中間言語 Stg から中間言語 Cmm へ変換する際に，新たなサンクを生成するコードになる．そのため，中間言語 Stg では，let 式を見つけるだけで，新たにサンクの生成を行う部位を特定できる．また，IS のデータ構成子適用が新たな IS を生成するコードになるため，IS のデータ構成子適用を見つけるだけで新たな IS の生成を行う部位を特定できる．

上書き可能なサンクは IS の引数として与えられているものであるため，上書き可能なサンクを見分けるためには，まず，let 式を見つけ，その変数が IS の引数に与えられているかどうかを確かめればよい．また，IS に関してはつねに上書き可能であるため，IS のデータ構成子適用を見つけるだけでよい．

このように，中間言語 Stg の段階では上書きする部位の特定が容易に行える．そのため本機構の実現では，中間言語 Stg の段階で上書き可能な部位の特定を行うようにした．特定した部位は構文木要素を変換しておき，Stg から Cmm へ変換する際に区別できるようにする．たとえば，図 12 (b) は図 12 (c) に変換する．

さらに，Stg から Cmm への変換系において，特定した部位に対して，上書きを行うコードを生成できるよう変更を加えた．ここで，上で説明したように Stg において上書き可能と特定した部位に対して，従来のコードも生成できるようにしておく．

Stg から Cmm への従来の変換系では，IS およびサンクの生成は 1 つのコードブロックとなり，割当てに十分なメモリが残っているかどうかのチェックをまとめて行う形で，**図 13** (a) のようなコードを生成していたのに対し，提案機構は図 13 (b) のようなコードを生成する．

```
  if (IS とサンクを生成するのにヒープが十分でない) then goto gc;
  サンクの生成
  IS の生成
  IS のリターン
gc:
  ごみ集めの開始
```

(a) 従来のコード

```
  if (IS 用更新フレームが積まれている) then goto thunk_check;
  if (IS とサンクを生成するのにヒープが十分でない) then goto gc;
  サンクの生成 (IS へのポインタを持たせる)
  IS の生成
  IS のリターン
thunk_check:
  if (サンクを上書きできる) then goto reuse;
  if (サンクを生成するのにヒープが十分でない) then goto gc;
  スタックから IS のアドレスを取り出す
  サンクの生成 (IS へのポインタを持たせる)
  IS の上書き
  上書きした IS のリターン
reuse:
  スタックから IS とサンクのアドレスを取り出す
  サンクの上書き
  IS の上書き
  上書きした IS のリターン
gc:
  ごみ集めの開始
```

(b) 提案機構で生成したコード

図 13　サンクの擬似コード
Fig. 13　Pseude code: Thunk.

5.2　実行時システムの変更

続いて，実行時システムに施した変更について述べる．

まず，IS 用更新フレーム（reuse_frame）を追加した．従来の更新フレームでは，評価結果への間接ポインタでサンクの上書き処理を行っていたが，提案機構ではサンクのコード自体に上書き処理が含まれるようになっているため，IS 用更新フレームでは，リメンバ

```
  if (IS が旧世代にない) then goto end;
  if (IS がすでにリメンバセットへ追加されている) then goto end;
  IS をリメンバセットへ追加
  IS の info pointer を付けかえる
end:
  IS 用更新フレームのポップ
  IS のリターン
```

図 14　IS 用更新フレームの擬似コード
Fig. 14　Pseudo code: reuse-frame.

セットに関する処理を行うだけでよい．

リメンバセットに関する処理では，リメンバセットへの追加が重複しないよう工夫する必要がある．提案機構では，同一の IS に対して複数回の更新が起こりうる．ゆえに，更新の起こった旧世代の IS をリメンバセットに単純に追加していくと，リメンバセットが膨大になり，ごみ集めの際に大きなオーバヘッドとなる．これを防ぐために，提案機構では，IS がすでにリメンバセットに追加されたか否かを容易に区別できるようにした．具体的には info pointer の付けかえによってその区別を可能にした．つまり，IS の info table を複製し，クロージャタイプのみを変更したものを用意しておき，これに付けかえる．このことによって，info pointer を付けかえたままでも，本来の IS と同じように計算が進められ，IS 用更新フレームにおいてはクロージャタイプを見ることで，すでにリメンバセットへ追加された IS かどうかが判断できる．IS 用更新フレームの擬似コードを**図 14** に示す．

また，Blackholing を行うため，ごみ集めの準備としてスタックを走査する関数に，IS 用更新フレームに対する分岐を追加した．3.3 節で述べたように，GHC 6.6.1 では info pointer を付けかえることで Blackholing を行っている．Blackholing が行われたサンクは，Blackholing 用の info table の情報を基に間接ポインタに必要な領域だけがコピーされる．これに対し提案機構では，サンクの領域すべてが必要である．しかし，サンクの大きさは後続計算により様々であるため，GHC 6.6.1 の方式をそのまま踏襲しようとすると，サンクのサイズごとに Blackholing 用 info table を用意しなければならない．そこで現在の実装では，info pointer の付けかえを行わず，IS 用更新フレームからたどれる IS およびサンクの保持するポインタをヌルポインタで上書きし，ヌルポインタに対する処理をごみ集め機構に追加することで Blackholing を実現している．

5.3　提案機構の効果

本節では，提案機構による得失について議論する．

まず，本機構がごみ集めに要する時間に与える影響について述べる．利点としては，メモリ消費量が抑えられることによるごみ集め回数の減少，その結果としてのごみ集め時間の削減があげられる．GHC では，オブジェクトが割り当てられる領域の大きさが実行中に変化しないため，メモリ消費量の減少は直接ごみ集めの回数，特に新世代ごみ集めの回数を減らすことにつながる．また，メモリ消費速度が遅くなることで，新世代ごみ集めの時間間隔が広がることにもなる．

一方，欠点としては，リメンバセットの増加があげられる．提案機構では，逆方向ポインタが従来よりも多く発生する．その結果，必然的にリメンバセットが大きくなり，新世代ごみ集めの際のルートが増加し，オーバヘッドとなる．

続いて，提案機構が計算時間（総実行時間からごみ集め時間を除いたもの）に与える影響について述べる．利点としては，オブジェクト割当て処理の減少があげられる．IS および後続計算のサンクに関して既存領域の上書きを行うため，新たにメモリ領域を割り当てる必要がなくなる．

一方，欠点としては，サンクのコードにおける分岐処理が従来よりも多くなることがあげられる（図 13）．また，IS 用更新フレームにおいてリメンバセットに追加する際も，すでに追加されたかどうかを見分けるための分岐が加わる．これらにより，従来よりも処理内容が若干増加することになり，オーバヘッドとなる．

ほかにも，新世代ごみ集めの間隔が広がることで，IS の後続計算以外のサンクを評価することで生成される間接ポインタが問題となることも考えられる．間接ポインタはごみ集めの際に解消されるため，ごみ集めの間隔が広がることで間接ポインタが長く残ることになる．そのため，間接ポインタを経由するアクセスが多い場合には，オーバヘッドとなりうる．

以上の考察から，提案機構のないオリジナルの GHC において，次のような性質を持つプログラムであれば，提案機構による効果を見込むことができると考えられる．

性質 1　総実行時間に対するごみ集め時間の割合が大きい．

さらに，性質 1 が満たされるという前提のもと，次の性質 2 も満足されれば，より大きな効果が期待できる．

性質 2　メモリ消費のうち，IS と後続計算のサンクによる消費が多い．特に，1 つの IS に関して次の近似値を多く求める（1 つの IS に関して求める近似値の列が長い）傾向がある．

6. 評　　価

5.3 節で述べた提案機構の効果を確認するため，実験を行った．実験環境は，Pentium 4 (3.0 GHz)，主記憶 1 GB の PC で，OS は Linux Kernel 2.6.8 である．

6.1 メモリ消費量・実行時間の測定

本節では，GHC 6.6.1 でコンパイルしたプログラム（IS はライブラリで提供）と，提案機構でコンパイルしたプログラム（IS は組み込みデータとして提供）の，メモリ消費量および実行時間の比較について述べる．ここで，メモリ消費量とは，プログラムがヒープ内に割り当てたメモリ量の累計を指す．

6.1.1 実 験 内 容

ナップサック問題，編集距離問題，8 パズルを解くプログラムにそれぞれ複数の入力を与え，実行時間およびメモリ消費量を測定した．入力は，ランダムに生成したものから実行時間が短くなりすぎないものを抜き出して，ナップサック問題と 8 パズルにはそれぞれ 100 個，編集距離問題には 54 個を与えた．

測定には，GHC でコンパイルされたプログラムに対する実行時オプション `-s` を利用した．コンパイルしたプログラムを `a.out` とすると，このオプションは以下のようにして用いる．

```
./a.out +RTS -sfilename -RTS
```

`+RTS ... -RTS` で囲まれた部分は，GHC でコンパイルしたプログラムに対する実行時オプションである．`-sfilename` は，`filename` に指定されたファイルに対して，プログラムの実行時の情報を出力するオプションである．実行時の情報には，メモリ消費量や実行時間が含まれる．

6.1.2 実験結果と考察

図 15 がナップサック問題，**図 16** が編集距離問題，**図 17** が 8 パズルの実験結果である．

各図とも，横軸は入力の番号を示している．図の (a) は，メモリ消費量の比（提案機構あり/なし），(b) は総実行時間の比（提案機構あり/なし）である．さらに，5.3 節における議論を確認するため，(c) に提案機構なしの場合の総実行時間に対するごみ集め時間の割合を，(d) に計算時間（総実行時間からごみ集めの時間を減じたもの）の比とごみ集め時間の比（いずれも，提案機構あり/なし）を示した．入力は，総実行時間の比によってソートを行ったのちに番号を振っており．(a)〜(d) とも入力の番号は共通している．

メモリ消費量に関しては，ナップサック問題（図 15 (a)）では，ばらつきはあるものの平均すれば 75%程度メモリ消費量が減少している．また，編集距離問題（図 16 (a)）および 8 パズル（図 17 (a)）では，ナップサック問題ほど減少率が大きくないものの，やはりメモリ消費量が減少していることを確認できる．具体的には，編集距離問題では平均して 15%程度，8 パズルでは平均して 3%程度の減少が見られた．

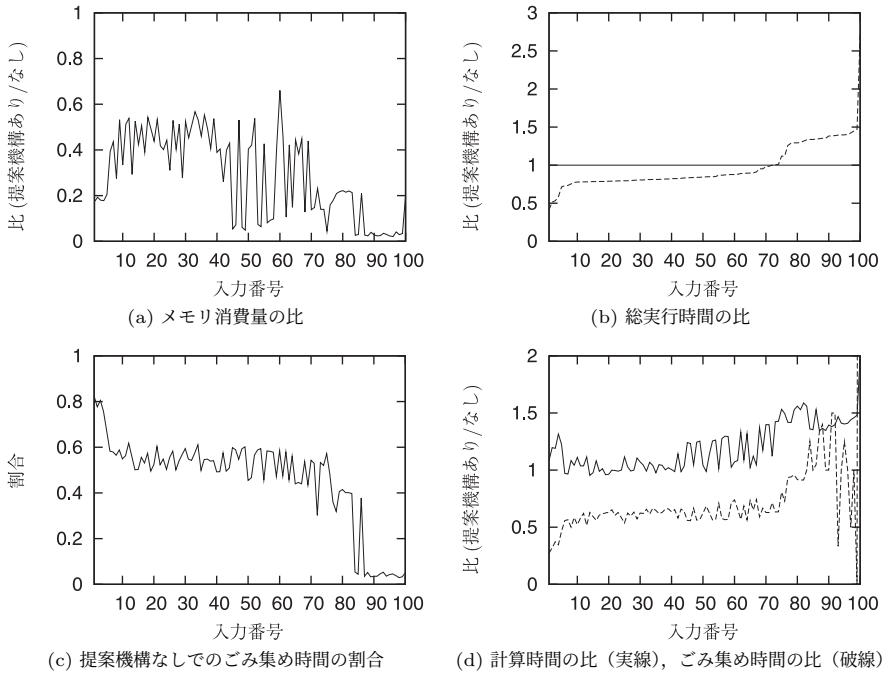
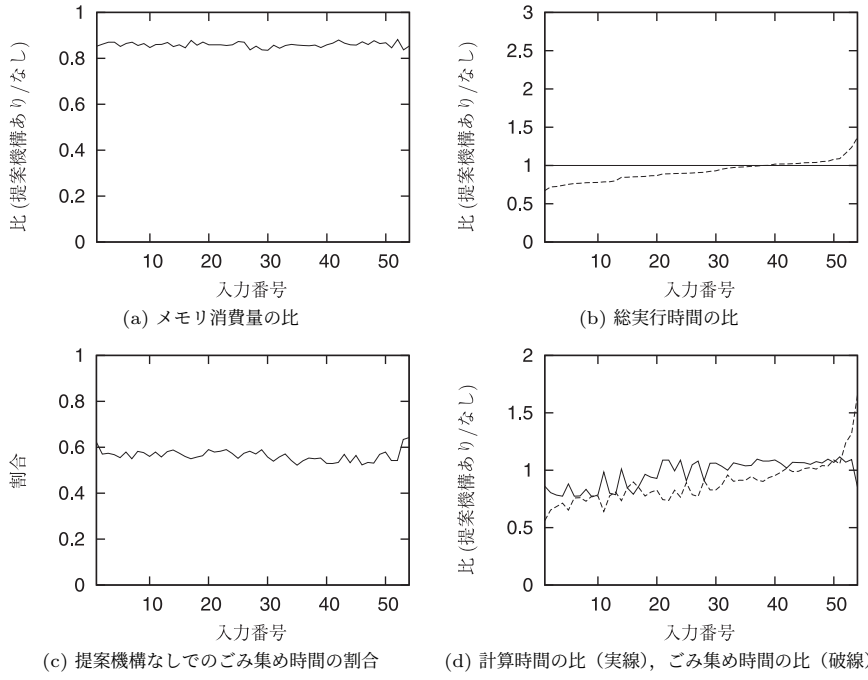


図 15　ナップサック問題
Fig. 15　Some results of Knapsack Problem.

総実行時間に関しては，ナップサック問題（図 15 (b)），編集距離問題（図 16 (b)）では，7 割程度の入力において総実行時間が減少した．また 8 パズル（図 17 (b)）においては，すべての入力で総実行時間が減少した．

ナップサック問題において図 15 (b) と (c) を比べると，提案機構の効果が大きいものは，総実行時間に対するごみ集め時間の割合が大きい（ほぼ 50%以上）場合，すなわち性質 1 が満足される場合であることが分かる．このとき (a) のメモリ消費量の比を見ると，提案機

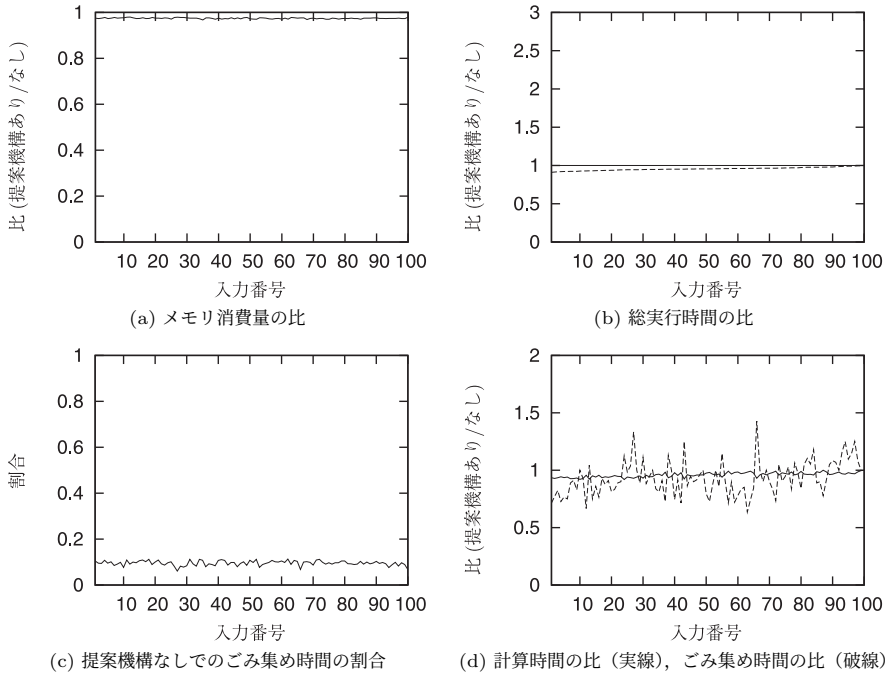


(a) メモリ消費量の比　(b) 総実行時間の比

(c) 提案機構なしでのごみ集め時間の割合　(d) 計算時間の比（実線），ごみ集め時間の比（破線）

図 16　編集距離問題

Fig. 16　Some results of Edit Distance Problem.



(a) メモリ消費量の比　(b) 総実行時間の比

(c) 提案機構なしでのごみ集め時間の割合　(d) 計算時間の比（実線），ごみ集め時間の比（破線）

図 17　8 パズル

Fig. 17　Some results of 8 puzzle.

構なしの場合のメモリ消費の多くが IS および後続計算のサンクによるものであり，それらが提案機構により削減されていること，すなわち性質 2 も満足されていることが分かる．その結果，ナップサック問題における 7 割程度の入力に関しては，提案手法による大きな効果が得られている．また，総実行時間が増加してしまい提案機構の効果が出ていない場合は，元々ごみ集め時間の割合が少ない，すなわち性質 1 がそもそも満たされていないという傾向も見られる．

編集距離問題では，図 16 (c) より，ナップサック問題で提案機構の効果が得られた場合と同様に，総実行時間に対するごみ集め時間の割合がおおむね 60%と高い，すなわち性質 1 を満足していると分かる．一方 (a) を見ると，提案機構によるメモリ消費量の減少は約 15%であり，性質 2 を満足する度合いはナップサック問題ほどではないということも見てとれる．その結果，総実行時間に関しては，一定の効果は見られるものの，ナップサック問題よりも効果が小さい傾向となっている．

8 パズルは，図 17 (c) から，総実行時間に対するごみ集め時間の割合が約 10%と，他の 2 つの問題よりかなり低い，すなわち性質 1 があまり満足されていないことが分かる．それに加え，(a) のメモリ消費量の減少率から，この問題ではそもそも IS および後続計算のサンクによって消費されるメモリの割合が少ない，つまり性質 2 もあまり満足されていない．総実行時間における提案機構の効果が他の 2 つの問題よりも小さいことは，以上の分析から説明できる．

以上の実験により，どの問題においても，IS および後続計算のサンクによって消費されるメモリ量を安定的に改善できていることが確認できた．また，図 15〜図 17 の (d) を見る

と，総実行時間が減少した入力に関しては，総実行時間からごみ集め時間を除いた純粋な計算時間は，問題によって傾向は異なるものの，提案機構の有無によって大きな違いは見られず，提案機構の効果はごみ集め時間の減少という形で現れている．実際，5.3 節で述べた性質 1 および 2 を満たす場合において提案機構の効果を見込めること，また，これらの性質を満たす度合いによって効果に違いがあることも確認できた．

一方，必ずしも総実行時間が改善されるとは限らないことも分かった．今回与えた入力では，ナップサック問題において 100 個のうち 28 個，編集距離において 54 個のうち 16 個の計 44 個に対し総実行時間の改悪が見られた．これらには，一般的に以下の傾向が見られる．

- 提案機構なしの場合の，総実行時間に対するごみ集めの時間の割合が小さい．
- 提案機構なしの場合よりも計算時間が増加している．

すなわち，総実行時間を支配する計算時間の増加がごみ集め時間の減少を打ち消し，逆効果として現れてしまっている．実際，ナップサック問題において，メモリ消費量が劇的に減少している場合においても総実行時間の増加が見られたケースは，これに該当している．しかしながら，改悪した 44 個の入力のうちの 11 個では，ごみ集め回数には減少が見られるもののごみ集め時間が増加していた．その原因は 5.3 節で述べた以外にもメモリの局所性などの要因も考えられるが，詳細な検討は今後の課題である．

6.2 コンパイル時間の測定

本節では，提案機構によるコンパイル時間への影響を調べるために行った実験について述べる．

6.2.1 実 験 内 容

提案機構を用いた場合と用いない場合のそれぞれで，コンパイル時間を計測した．プログラムには，6.1 節で使用したナップサック問題，編集距離，8 パズルを用いた．また，計測は 100 回行い，平均をとった．

6.2.2 実験結果と考察

表 1 に実験結果を示す．この結果より，コンパイル時間の増加は数%程度であり，提案機構による影響は比較的小さいことが確かめられた．

提案機構に由来するコンパイル時間増加の原因は，主に Stg から Stg への変換パスによるものと考えられる．このパスでは，構文木をひととおりたどりながら再構成するため，プログラムのサイズに比例した時間がかかる．しかし，GHC では，Core から Stg への変換，Stg から Cmm への変換，Core から Core および Stg から Stg の最適化のように構文木をひととおり変換する処理や，型の整合性チェック・実行時システムとのリンクなど，他の部分の処理に元々時間がかかるために，提案機構による影響が出にくいものと考えられる．

表 1　コンパイル時間の比較
Table 1　Compile time.

提案機構の有無	コンパイル時間（秒）（%）		
	ナップサック問題（53 行）	編集距離問題（66 行）	8 パズル（129 行）
なし	0.33（100%）	0.50（100%）	0.79（100%）
あり	0.35（106%）	0.52（104%）	0.81（102%）

7. おわりに

本論文では，遅延評価を基本とする純関数型言語の処理系における，定数領域しかメモリを消費しない IS の実装について述べた．この実装によって IS および後続計算のサンクによるメモリ消費量を安定的に改善できること，メモリ消費量を削減することで多くの場合において実行時間の改善が見込めることを示した．効果を見込めるのは，プログラムが 5.3 節で述べた性質 1，2 を満足する場合であり，実際これらの性質をどの程度満足するかにより，提案機構の効果に違いが出ることを，実験的に確認した．プログラムが性質 1 および 2 をどの程度満たすかは，具体的な入力データが与えられたうえでのプログラムの実行時の振舞いに依存する．そのため，提案機構による効果を静的に見積もることは難しい．

本研究では GHC 6.6.1 を用いたが，現時点での最新バージョンは 6.8 である．このバージョンでは，動的ポインタタギング[3] という手法が導入されている．この手法は，サンクやデータ構造を指すポインタの下位ビットにタグ付けをし，評価済みであるかどうか（サンクかデータ構造か）をポインタから判断することで実行効率を上げるというものである．動的ポインタタギングは，本論文の実装と併用が可能だと考えられる．そのため今後は，GHC 6.8 に提案機構を実装し，併用できることを確認する必要がある．

提案機構では，IS の近似値が枝刈りの判断にのみ用いられることを前提として IS とサンクの再利用を行ったが，近似値が枝刈りの判断にのみ用いられていることを，現在ではコンパイル時に静的に確認することはしていない．今後は，近似値が枝刈りの判断にのみ用いられていることを静的に確認する仕組みを開発することも必要である．具体的には，データフロー解析等によって近似値がどのように使われているか調べ，提案手法の適用の可否を判断することを考えている．

参 考 文 献

1) Iwasaki, H.: Pruning Unnecessary Computations using Improving Sequence, *Proc. 3rd Asian Workshop on Programming Languages and Systems* (*APLAS'02*), pp.46–57 (2002).
2) Jones, S.L.P.: Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine, *Journal of Functional Programming*, Vol.2, No.2, pp.127–202 (1992).
3) Marlow, S., Yakushev, A.R. and Jones, S.P.: Faster laziness using dynamic pointer tagging, *Proc. 2007 ACM SIGPLAN international conference on Functional programming* (*ICFP'07*), New York, NY, USA, ACM, pp.277–288 (2007).
4) Morimoto, T., Takano, Y. and Iwasaki, H.: Instantly Tuning a Naive Exhaustive Search into Three Efficient Searches with Pruning, *Proc. 9th International Symposium on Practical Aspects of Declarative Languages* (*PADL'07*), pp.65–79 (2007).
5) Richard, B.: *Introduction to Functional Programming using Haskell*, Prentice Hall (1998).
6) 高野保真，岩崎英哉：Improving Sequence を第一級の対象とする Scheme コンパイラ，第 8 回プログラミングおよびプログラミング言語ワークショップ（PPL 2006），pp.153–162 (2006).
7) 湯淺太一：Java アプリケーション組み込み用の Lisp ドライバ，情報処理学会論文誌：プログラミング，Vol.44, No.SIG 4(PRO17), pp.1–16 (2003).

(平成 20 年 2 月 17 日受付)
(平成 20 年 5 月 14 日採録)

田村　知博（学生会員）

1984 年生．2006 年電気通信大学電気通信学部情報工学科卒業．2008 年電気通信大学大学院電気通信学研究科情報工学専攻博士前期課程修了．2008 年 4 月より同研究科同専攻博士後期課程に在籍中．プログラミング言語とその処理系，特に関数型言語，遅延評価に興味を持つ．

高野　保真（学生会員）

1981 年生．2004 年電気通信大学電気通信学部情報工学科卒業．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博士前期課程修了．2006 年 4 月より同研究科同専攻博士後期課程に在籍中．プログラミング言語処理系の実装に興味を持つ．

岩崎　英哉（正会員）

1960 年生．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情報工学科教授．工学博士．記号処理言語，関数型言語，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．