要求駆動計算における要求粒度調節機構

森本武資[†] 岩崎英哉[††]

要求駆動計算において要求を細かく出せば，計算の必要性を細かい単位で調べられ，より多くの不要計算を除去できる．しかし，要求を細かくしすぎると，いずれ必要になる計算の必要性を調べるという無駄が生じ，せっかくの不要計算除去による改善効果を打ち消してしまう．これまでに，細かい要求を簡潔に記述するための言語機構が提案されているが，その機構において要求の細かさ（要求の粒度）を適切に調節する方法については十分な検討がなされていなかった．本論文では，この問題を解決するため，投機的評価を用いて要求の粒度を調節する機構を提案する．提案機構では，要求に応じて計算を進める際に，要求された計算の後続計算も投機的に進めることで，要求の粒度を調節する．適用例としてナップサック問題と 8 パズルを解くプログラムを取り上げ，実験により安定した改善効果を確かめた．

A Mechanism for Adjusting Granularity of Demand Driven Computation

Takeshi Morimoto[†] and Hideya Iwasaki[††]

Demand driven computation is potentially efficient by incrementally issuing small demands, because it helps prune many unnecessary computations by carefully checking whether each computation is necessary or not. However, if the granularity of demands are too small, the efficiency may be spoilt due to the overhead of the control of demand driven computation. Although a mechanism that enables programmers to concisely write small-demand driven programs has been already proposed, controlling granularity of demands is not well investigated. To resolve this problem, this paper proposes a mechanism that controls the granularity of demands by the introduction of speculative evaluation. In the proposed mechanism, when a small demand is issued, the rest computation for this demand is also evaluated speculatively, so that the granularity is adjusted. We described two concrete programs for knapsack problem and eight-puzzle with the proposed mechanism. Their experimental results show the stable effectiveness of the proposed mechanism.

1. はじめに

要求駆動型のプログラムは，要求に応じて必要な計算だけを進めるので，結果に寄与しない不要計算を除去することができる．そのため潜在的な実行効率が良く，遅延評価を行う関数型言語により簡潔に記述できる．要求を細かく出せば，計算の必要性を細かい単位で調べられ，より多くの不要計算を除去できることから，細かい要求を簡潔に記述するための言語機構[1)]が提案されている．この言語機構を用いれば，遅延評価における要求を細かくすることができ，探索問題を効率的に解くプログラムなどを簡潔に記述できる．

しかし，要求駆動にはオーバヘッドがともなう．必要以上に要求を細かくしすぎると，いずれ必要になる計算の必要性を調べるという無駄が生じ，せっかくの不要計算除去による改善効果を打ち消してしまう．この改善効果とオーバヘッドのトレードオフに対処するには，要求の細かさ（「要求の粒度」と呼ぶ）を適切に調節することが重要である．しかし，要求の細分化のみを行う従来の方法では，要求を適切な粒度に調節することができず，必要以上の要求の細分化によるオーバヘッドの増大が効率改善の妨げとなっていた．

本論文では，この問題を解決するため，要求の粒度を投機的評価を用いて調節する機構を提案する．提案する機構では，要求に応じて計算を進める際に，要求された計算の後続計算も投機的に進めることで，要求の粒度を調節する．

本論文の構成は以下のとおりである．まず 2 章で細

† 電気通信大学大学院情報工学専攻
Department of Computer Science, Graduate School of Electro-Communications, The University of Electro-Communications
†† 電気通信大学情報工学科
Department of Computer Science, The University of Electro-Communications

かい要求を簡潔に記述するための言語機構を説明したのち，3 章で細分化された要求駆動のオーバヘッドについて述べる．4 章で我々が提案する要求の粒度調節機構とその使用法を述べる．5 章で記述例としてナップサック問題と 8 パズルを解くプログラムを取り上げ，6 章で記述したプログラムの有効性を実験により検証する．7 章で考察を行い，8 章でまとめと今後の展望を述べる．

本論文では関数型言語 Haskell[2] に，要求の細分化を記述するための拡張を加えた記法を用いる．

2. Improving Sequence

細かい要求は Improving Sequence[1]（IS と略す）を用いることにより簡潔に記述できる．IS とは徐々に真の値に近づいていく近似値の列である．近似値を手がかりに列を要求駆動で読み進めることにより，真の値に至る途中の段階で，不要な後続計算を除去できる．IS を用いる計算では一度の要求駆動で次の近似値しか要求しないため，真の値に至るまでの計算を要求する通常の遅延評価よりも，要求の粒度が細かい．

2.1 IS の定義

IS は近似値，後続計算，二項関係からなるデータ構造である．後続計算もまた IS であるため，IS は列をなす．二項関係は列中の隣り合う近似値の関係を表明する．列を読み進むにつれ，近似値はこの二項関係において単調に改善されてゆき，真の値に近づいていく．この性質により，近似値と二項関係から，後続計算の必要性を判定できる．

IS の構成には，データ構成子 $\ll$ と零項構成子 ε を用いる．近似値 x，後続計算 xs，二項関係 R からなる IS は $x \ll_R xs$ と記述する．ε は計算の終端，すなわち真の値には後続計算がないことを意味する．たとえば次の記述

$$0 \ll_< 1 \ll_< 2 \ll_< 3 \ll_< \varepsilon$$

は，二項関係 $<$ のもとで，近似値が 0 から徐々に改善され真の値 3 に至る IS を表す．また，構文上の味つけとして $\gg$ がある．$\gg_R$ は $a\,R^{-1}\,b \equiv b\,R\,a$ なる R^{-1} に関する $\ll_{R^{-1}}$ と等しい．$\ll$ からなる IS を上昇列，$\gg$ からなる IS を下降列と呼ぶ．以後，読みやすさのために，二項関係の明示が不要な文脈においては，$\ll_R$，$\gg_R$ を単に $\ll$，$\gg$ と記す．

本論文では，IS の二項関係を，推移的かつ反対称的で全的な関係，すなわち

- aRb かつ bRc ならば aRc
- aRb かつ bRa ならば $a = b$
- すべての元について比較可能

を満たす関係に限定する．経験上，IS を用いるプログラムの多くはこの制約を満たすことを期待できる．

2.2 IS の構成法

IS を返す関数を記述するには，IS を用いない通常の関数定義に，計算の終端と近似値の記述を追加すればよい．リストの長さを返す関数 $length$ を例に，その記述法を説明する．以下は通常の $length$ の定義である．

$$\begin{aligned} &length\ xs = length'\ xs\ 0 \\ &length'\ []\ n = n \\ &length'\ (x : xs)\ n = length'\ xs\ (n+1) \end{aligned}$$

$length'$ の第 1 式は最終的な評価結果として n を返すことを意味し，第 2 式は後続計算が $length'\ xs\ (n+1)$ であることを意味している．そこで，まず第 1 式に計算の終端を示す $\ll \varepsilon$ を追加する．また，$length'\ xs\ (n+1)$ の結果が n より大きいことから，二項関係を $<$ とし，第 2 式に $n \ll$ を追加して，以下の関数定義を得る．

$$\begin{aligned} &length\ xs = length'\ xs\ 0 \\ &length'\ []\ n = n \ll \varepsilon \\ &length'\ (x : xs)\ n = n \ll length'\ xs\ (n+1) \end{aligned}$$

この $length$ に，たとえば $[9, 4]$ を与えれば $0 \ll 1 \ll 2 \ll \varepsilon$ という IS を返す．

この例から分かるように，IS を用いたプログラムは，IS を用いない素朴なプログラムの構造を保つ．この性質により，IS を用いれば，プログラムの簡潔さを失うことなく要求の細分化を記述できる．

2.3 IS の使用法

IS 上の関数を用いることにより，小さな要求による要求駆動を実現できる．主要な関数に min がある（**図 1**）．min は Burton が考案した $minimum$[3] を IS の記法に書き換えたものである．min は 2 つの上

$$\begin{aligned} &min\ \varepsilon\ ys = \varepsilon \\ &min\ xs\ \varepsilon = \varepsilon \\ &min\ (x \ll_R xs)\ (y \ll_R ys) \\ &\quad \mid x == y \quad = x \ll_R min\ xs\ ys \\ &\quad \mid x\,R\,y \quad = x \ll_R min\ xs\ (y \ll_R ys) \\ &\quad \mid otherwise = y \ll_R min\ (x \ll_R xs)\ ys \end{aligned}$$

$$\begin{aligned} &max\ \varepsilon\ ys = \varepsilon \\ &max\ xs\ \varepsilon = \varepsilon \\ &max\ (x \gg_R xs)\ (y \gg_R ys) \\ &\quad \mid x == y \quad = x \gg_R max\ xs\ ys \\ &\quad \mid y\,R\,x \quad = x \gg_R max\ xs\ (y \gg_R ys) \\ &\quad \mid otherwise = y \gg_R max\ (x \gg_R xs)\ ys \end{aligned}$$

図 1 min と max の定義
Fig. 1 Definitions of min and max.

昇列を引数にとり，最小値に至る上昇列を返す．近似値の小さい列だけを読み進めていき，どちらか一方の列が真の値に至った時点で min も計算を終えることにより，まだ真の値に到達していないもう一方の列の後続計算を除去する．min を用いた要求駆動の過程を以下に示す．

$$\begin{aligned} & min\ (1 \ll 2 \ll 4 \ll 6 \ll \varepsilon)\ (2 \ll 3 \ll \varepsilon) \\ \Rightarrow\ & 1 \ll min\ (2 \ll 4 \ll 6 \ll \varepsilon)\ (2 \ll 3 \ll \varepsilon) \\ \Rightarrow\ & 1 \ll 2 \ll min\ (4 \ll 6 \ll \varepsilon)\ (3 \ll \varepsilon) \\ \Rightarrow\ & 1 \ll 2 \ll 3 \ll min\ (4 \ll 6 \ll \varepsilon)\ \varepsilon \\ \Rightarrow\ & 1 \ll 2 \ll 3 \ll \varepsilon \end{aligned}$$

この例では，結果に寄与しない後続計算 $4 \ll 6 \ll \varepsilon$ を除去している．下降列に関して同様の処理を行う関数を max とする．

3. 細分化された要求駆動のオーバヘッド

関数プログラムは小さな関数の定義とそれらの組合せにより記述する．そのため関数プログラムには，たとえば $f_0\ (f_1\ (f_2\ x))$ というような連続する関数適用が頻繁に現れる．

IS を用いるプログラムでも関数適用は頻繁に現れる．たとえば f_i が IS を引数にとり IS を返す関数

$$f_i\ (x \ll xs) = g_i\ x \ll \cdots$$

であったとすると，その関数合成の評価結果は

$$f_0\ (f_1\ (f_2\ (x \ll xs)) \Rightarrow g_0\ (g_1\ (g_2\ x)) \ll \cdots$$

となる．x という 1 つの近似値に対し，$g_0\ (g_1\ (g_2\ x))$ という連続した関数適用がなされる．したがって要求の粒度を小さくし近似値が多く現れるようなプログラムでは，それだけ関数適用の回数が多くなる傾向がある．

もう 1 つ別の例を考える．

$$f\ x_i = a_i \ll g\ (f\ x_{i+1})$$

という形の再帰は，IS を用いるプログラムに現れる典型的なパターンである．ここで，関数 g は IS を引数にとり IS を返す関数である．簡単のため g を IS 上の次のような再帰的な関数

$$g\ (x \ll xs) = k\ x \ll g\ xs$$

とすると，関数 f を用いる計算は

$$\begin{aligned} f\ x_0 &\Rightarrow a_0 \ll g\ (f\ x_1) \\ &\Rightarrow a_0 \ll g\ (a_1 \ll g\ (f\ x_2)) \\ &\Rightarrow a_0 \ll k\ a_1 \ll g\ (g\ (f\ x_2)) \\ &\Rightarrow a_0 \ll k\ a_1 \ll g\ (g\ (a_2 \ll g\ (f\ x_3))) \\ &\Rightarrow a_0 \ll k\ a_1 \ll g\ (k\ a_2 \ll g\ (g\ (f\ x_3))) \\ &\Rightarrow a_0 \ll k\,a_1 \ll k\,(k\,a_2) \ll g\,(g\,(g\,(f\,x_3))) \end{aligned}$$

となる．i 番目の近似値 a_i を求めるのに，g を i 回連続で適用している．先ほどの例と同様，近似値を 1 つ求める際に複数の関数適用が現れるので，要求を細分化しすぎて近似値を細かく返すようにすると関数適用の回数が増えてしまう．

上述の g はかなり単純な関数であったが，2 つの IS を引数にとる実用的な関数でも同じことがいえる．たとえば min（図 1）は，典型的には次のような再帰関数の中で用いられる．

$$h\ x_i = a_i \ll min\ (h\ x_{2i})\ (h\ x_{2i+1})$$

h を用いる計算は，簡単のため $a_i = i$，二項関係を $<$ とすると，次のように進む．

$$\begin{aligned} h\ x_1 &\Rightarrow 1 \ll min\,(h\ x_2)\ (h\ x_3) \\ &\Rightarrow 1 \ll min\,(2 \ll min\,(h\ x_4)\ (h\ x_5)) \\ &\qquad\qquad (3 \ll min\,(h\ x_6)\ (h\ x_7)) \\ &\Rightarrow 1 \ll min\,(2 \ll min\,(4 \ll xs)\,(5 \ll ys)) \\ &\qquad\qquad (3 \ll zs) \end{aligned}$$

ここで，$min\ (h\ x_8)\ (h\ x_9)$，$min\ (h\ x_{10})\ (h\ x_{11})$，$min\ (h\ x_6)\ (h\ x_7)$ をそれぞれ xs，ys，zs とおいた．$1 \ll$ 以降の後続計算は，

$$\begin{aligned} & min\,(2 \ll min\,(4 \ll xs)\,(5 \ll ys))\,(3 \ll zs) \\ \Rightarrow\ & 2 \ll min\,(min\,(4 \ll xs)\,(5 \ll ys))\,(3 \ll zs) \\ \Rightarrow\ & 2 \ll min\,(4 \ll min\ xs\,(5 \ll ys))\,(3 \ll zs) \\ \Rightarrow\ & 2 \ll 3 \ll min\,(4 \ll min\ xs\,(5 \ll ys))\ zs \end{aligned}$$

のように進む．ここで近似値 4 に注目すると，この値が内側の min の第 1 引数の先頭の近似値から結果の近似値に浮上するまでに，min の適用を 2 回受ける必要がある．以上に見たように，細分化された要求駆動におけるオーバヘッドとは近似値を求めるために行う関数適用であり，近似値 1 つにつき，複数回の関数適用が行われる傾向がある．

4. 要求の粒度調節機構

4.1 近似値の間引き

前章で述べたオーバヘッドは，近似値を間引いて削減することができる．近似値を間引くのは，要求の粒度を粗くすることに相当する．そこで，本論文では，プログラム中に近似値を間引くことを陽に記述して，要求の粒度を調節する機構を提案する．

たとえば，$a_0 \ll a_1 \ll a_2 \ll \cdots$ から a_1 を間引くと $a_0 \ll a_2 \ll \cdots$ となる．すると，近似値 a_0 から一度の要求駆動で a_2 に至るまで計算を進めるので，a_1 を生成しそれを用いた計算を行うというオーバヘッドを減らせる．

近似値を間引くことは投機的評価を行うことに相当する．IS を用いる計算では近似値を手がかりに要求駆動で計算を進めるため，間引かなければ進めることのなかった不要計算を進めてしまうかもしれないという

リスクが間引きにはともなう．たとえば上の例においては，a_1 の間引きにより求めることとなった a_2 が，実は求める必要のない近似値かもしれないというリスクがある．

このため，近似値を適度に残すように間引きを行う必要がある．次の近似値がいずれ必要になりそうな近似値のみを間引くようにし，他の近似値は間引かず残すようにするのがよい．幸い，前章で見たように，近似値を 1 つ求める際には複数の関数適用が現れうるので，間引く近似値の個数が少なくても多くの関数適用を削減することを期待できる．

たとえば前章の h を用いる計算において，近似値 4 を間引いた場合，$2 \ll$ 以降の後続計算は次のようになる．

$$\begin{aligned} & min\ (min\ xs\ (5 \ll ys))\ (3 \ll zs) \\ \Rightarrow\ & min\ (min\ (8 \ll ws)\ (5 \ll ys))\ (3 \ll zs) \\ \Rightarrow\ & min\ (5 \ll min\ (8 \ll ws)\ ys)\ (3 \ll zs) \\ \Rightarrow\ & 3 \ll min\ (5 \ll min\ (8 \ll ws)\ ys)\ zs \end{aligned}$$

ここで，$8 \ll ws$ は 4 の後続計算 xs の評価結果である．この間引きにより，4 にともなうすべての min の適用を削減している．4 にともなう min の適用回数は少なくとも 2 回以上であり，文脈によってはより多くの回数になりうる．一方，この間引きには，xs が不要かもしれないというリスクがともなう．xs の評価内容は，

$$\begin{aligned} xs \Rightarrow\ & min\ (h\ x_8)\ (h\ x_9) \\ \Rightarrow\ & min\ (8 \ll us)\ (9 \ll vs) \\ \Rightarrow\ & 8 \ll min\ us\ (9 \ll vs) \\ \Rightarrow\ & 8 \ll ws \end{aligned}$$

であるから，xs の評価にかかるコストは h の適用が 2 回，min の適用が 1 回と一定である．ともなうリスクは一定であるのに，多くのオーバヘッドを削減することを期待できるため，xs がいずれ必要になりそうな文脈においては，4 を間引くことが有効である．

4.2 近似値の間引きの記述

近似値の間引きには，IS 上の関数 $spec$（**図 2**）を用いる．$spec\ p\ xs$ において，p は xs の先頭の近似値を間引くか否かを判断する真偽値である．p が $False$ のときは xs をそのまま返し，$True$ のときは xs の先頭の近似値を，それが最終値でなければ間引く．xs の 2 番目の近似値がいずれ必要になりそうならば $True$ となるように p を定めるとよい．$spec$ は IS を用いる既存の枠組みの中で記述できるため，新たな言語拡張を要さない．

$$\begin{aligned} & spec\ False\ xs = xs \\ & spec\ True\ (x \ll \varepsilon)\ = x \ll \varepsilon \\ & spec\ True\ (x \ll xs) = xs \end{aligned}$$

図 2 $spec$ の定義
Fig. 2 Definition of *spec*.

IS の先頭の近似値を間引いてもよいかどうか，すなわち 2 番目の近似値がいずれ必要になるかどうかは，その IS を生成する側でなく消費する側によって定まる．したがって，$spec$ は IS を消費する場所で使うのがよい．

たとえば，前章で取り上げた関数 f

$$f\ x_i = a_i \ll g\ (f\ x_{i+1})$$

によって生成される IS を間引く場合，

$$f\ x_i = spec\ p\ (a_i \ll g\ (f\ x_{i+1}))$$

と記述するよりも，

$$f\ x_i = a_i \ll g\ (spec\ p\ (f\ x_{i+1}))$$

と記述する方がよい．$f\ x_{i+1}$ の結果の IS は g でどのように消費されるかが分かるため，間引いてよいかどうかの判断材料が増えるためである．

$$f\ x_i = a_i \ll spec\ p\ (g\ (f\ x_{i+1}))$$

というように $g\ (f\ x_{i+1})$ の近似値を間引くのは，$f\ x_{i+1}$ の近似値を間引くよりも g の関数適用が増えてしまうので得策ではない．

同じことが前章の関数 h についてもいえる．h は次のように 2 つの IS を引数にとる関数であった．

$$h\ x_i = a_i \ll min\ (h\ x_{2i})\ (h\ x_{2i+1})$$

f 同様，a_i を間引くよりも，

$$\begin{aligned} h\ x_i = a_i \ll min\ & (spec\ p\ (h\ x_{2i})) \\ & (spec\ q\ (h\ x_{2i+1})) \end{aligned}$$

のように，$h\ x_{2i}$ と $h\ x_{2i+1}$ の近似値を間引く方がよい．$h\ x_{2i}$ と $h\ x_{2i+1}$ は min に渡され比較されることが明らかであるのに対し，a_i は何と比較されるか分からないからである．

$spec$ の第 1 引数に与える判定式の定めかたを，上の h のように，min の両引数に $spec$ が用いられている場合，すなわち

$$min\ (spec\ p\ xs)\ (spec\ q\ ys)$$

という形をしている場合を例にとって説明する．ここで，xs と ys は以下のような IS を生成するものとする．

$$\begin{aligned} & xs = x \ll x' \ll xs' \\ & ys = y \ll y' \ll ys' \end{aligned}$$

p と q の定めかたを考えるため，$spec$ が用いられていない場合，すなわち $min\ xs\ ys$ を考える．$min\ xs\ ys$ が生成する IS は，次のいずれかになる．

(a) $x \ll x' \ll \cdots$
(b) $y \ll y' \ll \cdots$

(c) $x \ll y \ll \cdots$
(d) $y \ll x \ll \cdots$

もしも $min\ xs\ ys$ の結果の IS，すなわち上の IS の 2 番目の近似値まで必要とされるならば，(a) の場合には xs の最初の近似値 x を，(b) の場合には ys の最初の近似値 y を間引くようにするのが有効であろう．なぜなら，いずれ 2 番目の近似値（(a) の場合には x'，(b) の場合には y'）を必要とするので，最初の近似値 x あるいは y を用いた計算がオーバヘッドになってしまうためである．

実際，$min\ xs\ ys$ の結果の IS の近似値を 2 番目まで必要とするような場面は，単純なプログラムでない限り，きわめて多いと考えられる．なぜなら，h のように min を再帰的に用いる計算は探索問題を表しているからである．ほとんどの探索問題においては探索する空間が爆発的に広がるので，探索空間の奥深くにある解を得るまでの間に，min の結果の IS を深く読み進める必要が生じることが多い．

以上の考察より p と q は次のように定めるのがよい．

p = (a) となるための条件，すなわち
　$x' \le y$ となるための条件
q = (b) となるための条件，すなわち
　$y' \le x$ となるための条件

一般的には，IS の二項関係を R とすると，この条件は次のように表される．

$p = x'\ R\ y$ または $x' == y$ となるための条件
$q = y'\ R\ x$ または $y' == x$ となるための条件

同様に $max\ (spec\ p\ xs)\ (spec\ q\ ys)$ の場合には，

$p = y\ R\ x'$ または $x' == y$ となるための条件
$q = x\ R\ y'$ または $y' == x$ となるための条件

とすればよい．

5. 記 述 例

ナップサック問題と 8 パズルを解くプログラムを例題にとり，提案機構の有効性を検証する．二者択一のナップサック問題と多者択一の 8 パズルとでは max や min の適用方法が異なるため，間引きの判定式の導きかたが異なる．

5.1 ナップサック問題

ナップサック問題とは，数種類の品物をナップサックの容量を超えないように詰め合わせて得られる，最大の総利得を求める問題である．入力には，ナップサックの容量と，ナップサックに入れる品物のリストが与えられる．品物はそれぞれ大きさと利得が異なり，同じ品物をいくつでも詰めることができる．簡単のため，品物のリストは大きさあたりの利得（利得率と呼ぶ）の大きい順に整列してあるものとする．

```
ks cap [] sum = sum
ks cap objs@((val, size) : rest) sum
  | cap == 0 = sum
  | cap  <  0 = −∞
  | cap  >  0 =
    maxi (ks (cap − size) objs (sum + val))
         (ks  cap          rest  sum)
```

a. 素朴な記述

```
ks cap [] sum = sum ≫ ε
ks cap objs@((val, size) : rest) sum
  | cap == 0 = sum ≫ ε
  | cap  <  0 = −∞ ≫ ε
  | cap  >  0 = sum + (val/size) ∗ cap ≫
    max (ks (cap − size) objs (sum + val))
        (ks  cap          rest  sum)
```

b. IS を用いる記述

```
ks cap [] sum = sum ≫ ε
ks cap objs@((val, size) : rest) sum
  | cap == 0 = sum ≫ ε
  | cap  <  0 = −∞ ≫ ε
  | cap  >  0 = sum + (val/size) ∗ cap ≫
    max′ (ks (cap − size) objs (sum + val))
         (ks  cap          rest  sum)
    where
    max′ xs ys=max (spec (cap ≥ size ∗ 2) xs)
                   (spec (cap < size)     ys)
```

c. 粒度を調節する記述

図 3 ナップサック問題のプログラム
Fig. 3 Programs for Knapsack problem.

まず，IS を用いない素朴なプログラムを示す（**図 3** a）．cap はナップサックの容量，$objs$ は品物のリスト，sum は総利得を示す累積変数である．個々の品物は利得 val と大きさ $size$ の組として表現される．第 1 式は品物がない場合の記述である．第 2 式の $cap == 0$ は空きがないことに対応し，$cap < 0$ は実現不可能であることに対応する．$maxi$ は 2 つの数を引数にとり，その最大値を返す関数である．$maxi$ の第 1 引数は $objs$ の先頭の品物をさらに 1 つ詰めた場合の総利得であり，第 2 引数はその品物をそれ以上詰めない場合の総利得である．

次に，IS を用いるプログラムを示す（図 3 b）．この関数 ks は，二項関係を $\le$ とする下降列を返す．こ

れは素朴なプログラムに計算の終端と近似値の記述を追加しただけである．近似値には利得率が最大の品物を隙間なく詰めた，いわば理想的な総利得を用いる．品物のリストは利得率の大きい順に並んでいるので，この値は，先頭の品物の利得率とナップサックの容量の積として表せる．この値とこれまでの総利得を加えた値を近似値とする．

提案手法に基づき，粒度調節の記述を追加する．前章での議論に基づき，$max\,(x \gg x' \gg xs')\,(y \gg y' \gg ys')$ において，$x' \geq y$ のとき x を間引き，$y' \geq x$ のとき y を間引くようにすればよい．ここでは，間引きの判定式を，プログラムを展開することにより求める．

ks における max の 2 つの引数は，

$x \gg x' \gg xs' = ks\ (cap-size)\ objs\ (sum+val)$

$y \gg y' \gg ys' = ks\ \ cap\qquad\quad rest\ \ sum$

であるから，$cap - size \geq 0$ のとき，

$$\begin{aligned} &x \gg x' \gg xs' = \\ &\quad (sum+val) + (val/size) * (cap-size) \gg \\ &\quad max\ (ks\ (cap-size*2)\ objs\ (sum+val*2)) \\ &\qquad\quad (ks\ (cap-size)\quad rest\ (sum+val)) \end{aligned}$$

$$y = \begin{cases} sum, & if\ rest == [\,] \\ sum+(v/s)*cap, & if\ rest == ((v,s):r) \end{cases}$$

となる．ゆえに，

$$\begin{aligned} x &= (sum+val) + (val/size) * (cap-size) \\ &= sum + (val/size) * cap \\ y &\leq sum + (v/s) * cap \\ &\leq x \end{aligned}$$

が成り立つ（$objs$ は利得率で降順に整列してあるから $v/s \leq val/size$ である）．さらに，$cap-size*2 \geq 0$ のとき，

$x' = (sum+val*2)+(val/size)*(cap-size*2)$

となり，$x' = x \geq y$ が成り立つ．したがって，$cap \geq size * 2$ のとき x を間引くようにすればよい．

また，$cap - size < 0$ のとき，$x = -\infty$ となるから，$y' \geq x$ である．ゆえに，$cap < size$ のとき y を間引くようにすればよい．IS を用いる従来の記述に，判定式 $cap \geq size * 2$ と $cap < size$ による間引きの記述を追加して，粒度を調節する記述（図 3 c）を得る．

粒度を調節する記述は簡潔である．素朴な記述との類似性に注意されたい．粒度を調節するプログラムは，素朴なプログラムの構造を完全に残している．追加した記述は，本質的には，計算の終端と，近似値，間引きの記述のみである．このように，提案手法に基づけば，簡潔さを失うことなく粒度の調節を記述できる．

	1	2
3	4	5
6	7	8

図 4 8 パズルの目的配置
Fig. 4 Goal state of Eight-puzzle.

```
pz state cost
  | is_goal state = cost
  | otherwise     =
    foldl1 mini [pz s (cost + 1) | s ← kids state]
```

a. 素朴な記述

```
pz state cost
  | is_goal state = cost ≪ ε
  | otherwise     = cost + sum_md state ≪
    foldl1 min [pz s (cost + 1) | s ← kids state]
```

b. IS を用いる記述

```
pz state cost
  | is_goal state = cost ≪ ε
  | otherwise     = cost + sum_md state ≪
    foldl1 min' [pz s (cost + 1) | s ← kids state]
    where
    min' (x ≪ xs) (y ≪ ys)
         = min (spec (x + 2 ≤ y) (x ≪ xs))
               (spec (y + 2 ≤ x) (y ≪ ys))
```

c. 粒度を調節する記述

図 5 8 パズルのプログラム
Fig. 5 Programs for Eight-puzzle.

5.2 8 パズル

8 パズルとは，3 × 3 に仕切られた盤上にある，1〜8 と書かれた 8 つのタイルを，**図 4** のような配置に揃えるパズルである．ここでは解に至る最小の手数を求めるプログラムを取り上げる．

まず，IS を用いない素朴なプログラムを示す（**図 5** a）．タイルが目的の配置になったら手数を返して計算を終える．揃っていなければ，タイルを 1 つスライドした配置から解に至る手数を再帰的に求めて，それらの最小値をとる．$state$ はタイルの配置，$cost$ は手数を示す累積変数である．is_goal は配置を判定し，$kids$ は空白に隣接する各タイルをスライドした各配置をリストで返す．$foldl1\ f\ xs$ はリスト xs の要素を関数 f で結合し，$mini\ x\ y$ は 2 つの整数 x，y の最小値を返す．$[f\ x \mid x \leftarrow xs]$ はリスト xs の各要素に関数 f を適用した値のリストを返す．

次に，IS を用いるプログラムを示す（図 5 b）．この

関数 pz は，二項関係を $\leq$ とする上昇列を返す．近似値にはマンハッタン距離を用いる．マンハッタン距離 md は，各座標の差の絶対値の和

$$md\ (x_1, y_1)\ (x_2, y_2) = |x_1 - x_2| + |y_1 - y_2|$$

と表される．あるタイルの現在地から目的地までのマンハッタン距離は，そのタイルを目的地にスライドさせるのにかかる最小の手数である．ゆえに，現在の配置から目的の配置に至るまでにかかる最小の手数は，各タイルの目的地までのマンハッタン距離の総和として表せる．この総和をこれまでの手数に加えた値を近似値とする．

提案手法に基づき，粒度調節の記述を追加する．$min\,(x \ll x' \ll xs')\,(y \ll y' \ll ys')$ において，$x' \leq y$ のとき x を，$y' \leq x$ のとき y を間引くようにすればよい．間引きの判定式を，ks とは異なる方法により求める．pz では min を可変長のリストに適用しているので，ks のようにはプログラムを展開できない．代わりに，pz が構成する上昇列の性質を利用する．

pz は，隣り合う 2 つの近似値 a，a' の差が 0 か 2 となる上昇列 $a \ll a' \ll as'$ を構成する．なぜなら，8 パズルにおいては，1 手で動かせるタイルは 1 つであり，動かしたタイルは目的地に近づくか，離れるか，どちらかになるからである．動かすのに 1 手（+1）かかるので，近づく（−1）と近似値は変わらず，離れる（+1）と 2 つ増える．pz が構成した上昇列は min に渡されるが，min の引数はすべて同じ配置の pz から派生した上昇列なので，min が構成する上昇列においても，隣り合う近似値の差は 0 か 2 となる．したがって，pz が構成するすべての上昇列において $a' \leq a+2$ が成り立つ．

pz のこの性質により，$min\ (x \ll x' \ll xs')\ (y \ll y' \ll ys')$ において，$x' \leq x+2$ となるから，$x+2 \leq y$ であれば $x' \leq y$ が成り立つ．ゆえに，$x+2 \leq y$ のとき x を，$y+2 \leq x$ のとき y を間引くようにする．IS を用いる従来の記述に，この判定式による間引きの記述を追加して，粒度を調節する記述（図 5 c）を得る．

6. 実　　験

粒度調節の効果を調べるために実験を行った．ナップサック問題を解く ks と 8 パズルを解く pz について，

- 粒度を調節しないプログラム BASE（図 3b, 図 5b）
- 粒度を調節するプログラム SPEC（図 3 c，図 5 c）

を実行した．プログラムには複数の入力を与え，入力ごとに個別に実行し計測した．ks には文献 4) と同様の方法で無作為に選んだ入力を 155 個与えた．pz の入力には無作為に選んだ初期配置を 831 個与えた．

実験には，筆者らが開発した Scheme 風の処理系を用いた．これは，遅延評価を行い，IS を言語プリミティブに備えるように，Java アプリケーション組み込み用の Lisp ドライバ[5] を拡張した，インタプリタに基づく処理系である．実験では，コンパイラに近い性能を得るために，min などの主要な関数に加え，ks や pz など，実行にかかわるすべての関数を組み込み関数として実装した．使用した Java ランタイムは Sun JDK 1.5 であり，オプション `-Xmx768m` を指定した．実行時間の計測には，ヒープサイズを変えないように JIT を行ったうえで，`com.sun.management` パッケージの `OperatingSystemMXBean.getProcessCpuTime()` と `GarbageCollectorMXBean.getCollectionTime()` を用いた．Linux 2.4.26 を OS として，Pentium4 2.8 GHz と 1 GB のメモリを備えた計算機の上で実行した．

計測結果を**図 6** と**図 7** に示す．グラフの横軸はプログラムに与えた個々の入力であり，BASE に対する SPEC の実行時間の比率が昇順になるよう整列している．また，両グラフの代表的な入力 A〜H における計測結果の詳細を**表 1** と**表 2** に示す．

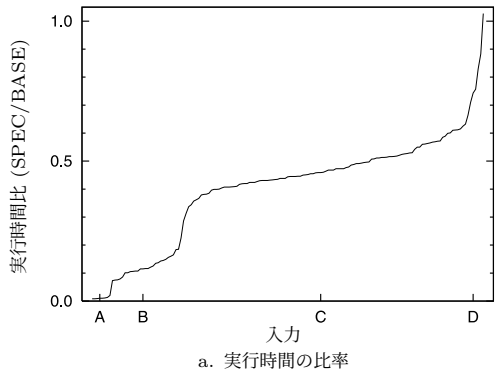
6.1 実 行 時 間

図 6 a と図 7 a は BASE の実行時間に対する SPEC の実行時間の比率を示すグラフである．ここでは CPU 時間と GC 時間の合計を実行時間とした．SPEC の実行時間はおおむね BASE の 60〜70%以下であった．いくつかの入力では BASE の 10%以下になるという大きな改善が得られた．改悪した入力は ks において 1 つだけあったが，その超過時間は BASE の 3%だけであった．この結果により，粒度調節により実行時間を安定的に短縮できることが示された．

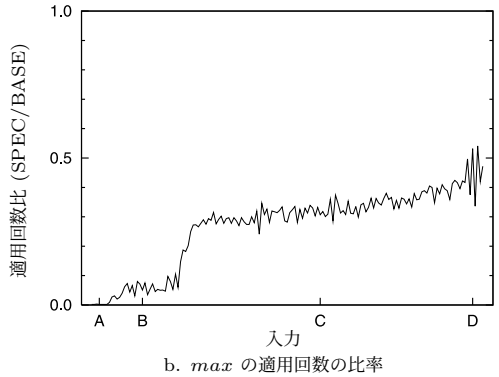
6.2 max，min の適用回数

図 6 b と図 7 b は，BASE の max，min の適用回数に対する，SPEC の max，min の適用回数の比率を示すグラフである．max，min の適用回数は実行時間にほぼ比例している．この結果は max，min の関数適用という，細分化された要求駆動におけるオーバヘッドを安定的に削減できたことが，実行時間の安定的な短縮をもたらしたことを示している．

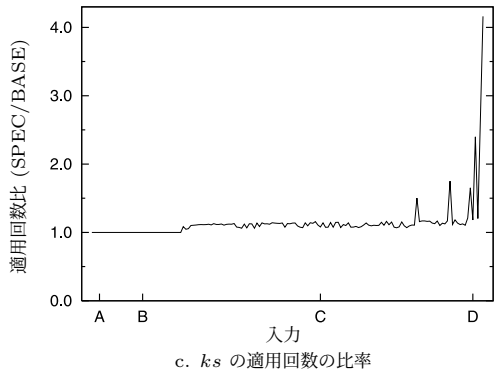
SPEC で間引いた近似値の個数と，削減された max，min の適用回数を調べると，近似値を 1 つ間引くごとに，平均的に，ks では 8 回の max の適用を削減でき，pz では 3 回弱の min の適用を削減できたことが分かった．このように，近似値の間引きが関数適用回数の削減に役立ち，結果として実行時間の削減に寄与することが示された．



a. 実行時間の比率

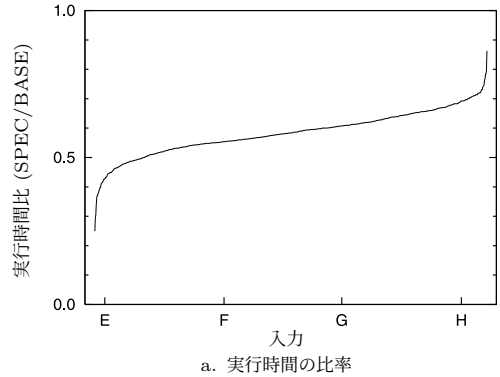


b. max の適用回数の比率

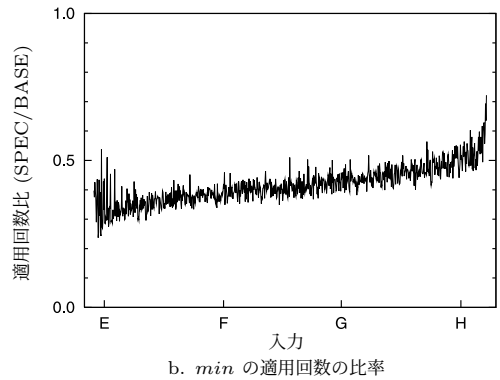


c. ks の適用回数の比率

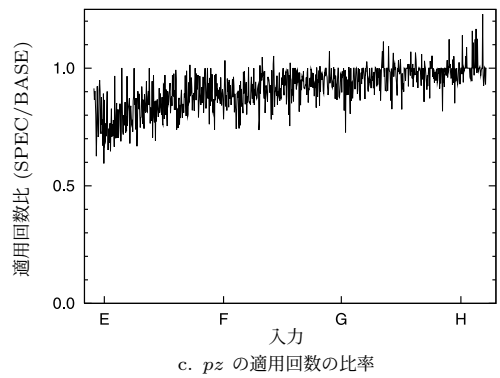
図 6 ks の計測結果
Fig. 6 Results of ks.



a. 実行時間の比率



b. min の適用回数の比率



c. pz の適用回数の比率

図 7 pz の計測結果
Fig. 7 Results of pz.

6.3 ks，pz の適用回数

図 6 c と図 7 c は，BASE の ks，pz の適用回数に対する，SPEC の ks，pz の適用回数の比率を示すグラフである．BASE は必要最小限の回数しか ks，pz を呼んでいないので，SPEC と BASE の差は，間引きにより進めてしまった不要計算の回数である．

ks で，間引きによる不要計算の増大がない場合には，実行時間が 2%未満と，間引きの効果が高い場合があった．また，多くの場合は不要計算の増大が 5〜15%程度であり，実行時間は 50%以下と効果が出ている．また，不要計算が大幅に増大しても実行時間としては一定の効果があることが示された．ks における不要計算の増大の最大値は 316%であったが，実行時間の改悪が 3%で済んでいるのは注目に値する．

pz では，多くの入力で SPEC の適用回数が BASE を下回った．その理由は，8 パズルが複数の最小解を持つことが多い[6] ためだと考えられる．BASE が複数の解に同時に近づくように計算を進めるのに対して，SPEC は 1 つの解に素早く到達し他の解に至る計算を枝刈りできたために，BASE よりも少ない回数で計算

表 1 代表的な入力における ks の計測結果
Table 1 Results of ks at representative points.

代表的な入力	実行時間（sec）			max の適用回数			ks の適用回数		
	BASE	SPEC	比	BASE	SPEC	比	BASE	SPEC	比
A	54.2	0.6	0.01	3,118,752	7,448	0.002	4,993	4,993	1.00
B	8.6	1.0	0.12	1,002,613	50,649	0.05	48,239	48,239	1.00
C	31.7	14.6	0.46	2,980,757	916,335	0.31	582,871	629,307	1.08
D	25.6	19.0	0.74	2,050,075	1,091,401	0.53	591,145	698,547	1.18

表 2 代表的な入力における pz の計測結果
Table 2 Results of pz at representative points.

代表的な入力	実行時間（sec）			min の適用回数			pz の適用回数		
	BASE	SPEC	比	BASE	SPEC	比	BASE	SPEC	比
E	10.7	4.5	0.43	640,180	213,291	0.33	433,699	258,087	0.60
F	25.9	14.3	0.55	1,635,867	687,118	0.42	1,081,618	847,198	0.78
G	12.3	7.5	0.61	808,071	369,992	0.46	486,599	472,239	0.97
H	5.8	4.0	0.69	381,933	192,028	0.50	236,679	236,672	1.00

を終えることができた．pz における不要計算の増大の最大値は 23%であり，その入力での実行時間の改善は 29%であった．

7. 考　　察

7.1 要求駆動における投機的評価

本研究は Optimistic Evaluation [7] に着想を得ている．Optimistic Evaluation とは「要求駆動計算の大半の部分計算は結局は必要になる」という経験則に基づき，投機的評価によって，通常の遅延評価のオーバヘッドを削減する評価戦略である．経験則に従えば「今のところ必要でない計算であってもいずれ必要になる可能性が高い」ので，まだ必要ではない計算をも投機的に評価することによって，計算の遅延にかかる（thunk の生成などの）オーバヘッドを削減する．投機の判断に実行時の情報を利用することで，静的な正格性解析[8] では行えないような，実行内容に即した最適化を行うことができる．

我々の提案手法においても，細分化された要求駆動のオーバヘッドを削減するために，同様の経験則に基づき，かつ，(min の周辺の変数値や近似値といった）実行時の情報を用いた投機的評価を行っている．細分化された要求駆動には投機的評価にともなうリスクが少ないという特徴があるため，Optimistic Evaluation のような大がかりな機構を用いなくても，$spec$ という単純な関数を用いるだけで安定的にオーバヘッドを削減することができた．

7.2 粒度調節と探索戦略

要求の粒度を調節することにより，計算全体としては最良優先的に探索しつつも，状況に応じて局所的に深さ優先的に探索するような探索戦略を実現できる．

探索問題を最良優先的に解くプログラムは，粒度を調節するまでもなく，IS を用いるだけで簡潔に記述できる．たとえば 3 章の関数 h

$$h\ x_i = a_i \ll min\ (h\ x_{2i})\ (h\ x_{2i+1})$$

を用いる計算 $h\ x_1$ は，

$$\begin{aligned} a_1 \ll min\ (a_2 \ll min\ (a_4 \ll min\ (a_8 &\ll \cdots) \\ &(a_9 \ll \cdots)) \\ (a_5 \ll min\ (a_{10} &\ll \cdots) \\ &(a_{11} \ll \cdots))) \\ (a_3 \ll min\ (a_6 \ll min\ (a_{12} &\ll \cdots) \\ &(a_{13} \ll \cdots)) \\ (a_7 \ll min\ (a_{14} &\ll \cdots) \\ &(a_{15} \ll \cdots))) \end{aligned}$$

となり，近似値を手がかりに最良優先的に最小値を求める探索問題を解く．すなわち，計算の進行過程の各時点において，最良（最小）の近似値を持つ後続計算だけを進めることにより，不要計算を進めることなく最小値を求める．

一方，粒度調節により，たとえば近似値 a_2，a_4，a_8 を間引いた場合には，

$$\begin{aligned} a_1 \ll min\ (min\ (min\ as \\ &(a_9 \ll \cdots)) \\ &(a_5 \ll \cdots)) \\ &(a_3 \ll \cdots) \end{aligned}$$

というように，局所的に深さ優先的な探索を行うようになる．すなわち，投機的に進めた計算の内部（$a_4 \ll \cdots$）でさらに投機がなされる（a_4 を間引く）といった投機の連鎖が起こりうる．このように投機が局所的に連鎖することによって，計算全体としては最良優先

的に探索しつつも，状況に応じて局所的に深さ優先的に探索することができる．

spec を用いるプログラムは *spec* を用いない元のプログラムの構造を完全に保つから，提案手法を用いれば，このような複雑な計算を，*spec* という単純な関数を用いるだけで，簡潔さを失うことなく記述できる．

8. ま と め

本論文では，近似値を間引くことで，細分化された要求駆動のオーバヘッドを安定的に削減できることを示した．また，構造の異なる 2 つのプログラムを例にとり，提案機構の効果的な使用法を示した．現段階では，間引きの判定式をプログラマが陽に与える必要があるが，*ks* のように *max* を二者択一で用いるようなプログラムであれば，プログラムの展開という半ば機械的な作業により判定式を導くことができるという知見が得られた．今後は，この知見に基づいた最適化機構を，IS を第 1 級の対象とするコンパイラ[9] に実装する予定である．

参 考 文 献

1) Iwasaki, H.: Pruning Unnecessary Computations using Improving Sequences, *Proc. 3rd Asian Workshop on Programming Languages and Systems*, pp.46–57 (2002).
2) Bird, R.: *Introduction to Functional Programming using Haskell*, Prentice Hall (1998).
3) Burton, F.W.: Encapsulating Non-determinacy in an Abstract Data Type with Determinate Semantics, *J. Functional Programming*, Vol.1, No.1, pp.3–20 (1991).
4) Sahni, S.: Approximate Algorithms for the 0/1 Knapsack Problem, *J. ACM*, Vol.22, No.1, pp.115–124 (1975).
5) 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.SIG4 (PRO17), pp.1–16 (2003).
6) Reinefeld, A.: Complete Solution of the Eight-Puzzle and the Benefit of Node Ordering in IDA*, *Proc. 13th Intl. Joint Conf. on Artificial Intelligence* (*IJCAI'93*), pp. 248–253 (1993).
7) Ennals, R. and Peyton Jones, S.L.: Optimistic Evaluation: an Adaptive Evaluation Strategy for Non-strict Programs, *Proc. Intl. Conf. on Functional Programming* (*ICFP'03*), pp.287–298 (2003).
8) Wadler, P. and Hughes, R.J.M.: Projections for Strictness Analysis, *Functional Programming Languages and Computer Architecture* (*FPCA'87*), Kahn, G. (Ed.), LNCS 274, pp.385–407 (1987).
9) 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'06*) 論文集，pp.153–162 (2006).

(平成 18 年 6 月 26 日受付)
(平成 18 年 9 月 14 日採録)

森本 武資（学生会員）

1978 年生．2001 年電気通信大学情報工学科卒業．2003 年同大学大学院情報工学専攻博士前期課程修了．現在，同専攻博士後期課程在学中．関数型言語の研究に従事．日本ソフトウェア科学会会員．

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 年同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，東京大学大学院工学系研究科情報工学専攻，電気通信大学情報工学科助教授を経て，2004 年 1 月から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．