関数型言語処理系におけるデータ構成子の unbox 化

神門 有史　田中 哲朗　武市 正人

遅延評価型の関数型言語は，理論的な取扱いが容易でプログラムも簡潔に書ける点が評価されているが，実行効率の問題が指摘されてきた．しかし，最近になって Glasgow Haskell Compiler に代表される効率的な言語処理系が登場し，実用的なプログラミング言語としても注目を集めている．

関数型言語処理系の効率的な実装を可能にした技術が大域的なプログラム解析をもとにした最適化であるが，この 1 つが正格性解析を用いた引数の unbox 化によるメモリアクセスの軽減である．本論文では，この最適化をさらに進めて，unbox 化を関数引数だけでなく代数データ型のデータ構成子の要素にも適用する方法を提案する．この方法に基づき，Glasgow Haskell Compiler のコード生成部を改良し，いくつかのプログラムに対して実行効率の向上を確認した．

1 はじめに

Haskell に代表される遅延評価に基づく関数型言語は，無限リストの利用やデータと制御の分離によるモジュール性の向上など，ML や Scheme などの先行評価に基づく関数型言語よりもプログラムの記述性は高いが，その効率的な処理系の実現は困難とされてきた．

Unboxing Data Constructors in a Lazy Functional Language.
Yuuji Kando, Masato Takeichi, 東京大学大学院工学系研究科, School of Engineering, University of Tokyo.
Tetsuro Tanaka, 東京大学教育用計算機センター, Educational Computer Centre, University of Tokyo.
コンピュータソフトウェア, Vol.14, No.4 (1997), pp.70–75.
[小論文] 1996 年 8 月 16 日受付.

しかし，大域的解析に基づく正格性解析 (strictness analysis) を用いた関数引数の unbox 化 などによって，その差は縮まりつつある．それでも，遅延評価の枠組では代数データ型のデータ構成子を unbox 化 するのは難しく，リストなどの構造のあるデータを効率良く表現するのが課題の 1 つとなっている．

遅延評価の枠組の中では代数データ型に関しては，その要素をポインタで表す必要がある (図 1(a))．遅延評価では要素として未評価の値 (サンク) が入れられ，評価後に弱頭部正規形で上書き (update) されるからである．

これに対して，要素にポインタを使わない実現 (unboxed value)，すなわち，整数自体を格納できれば効率が良い (図 1(b))．Peyton-Jones らは，プログラマが明示的に unboxed value を指定する方法を提案した [6]．この方法では，unboxed Int 型である `Int#` をもつ新たなデータ型:

```
data IList = INil | ICons Int# IList
```

を定義し，対応する関数を書き換える．さらに Hall らは，プログラマの手間を軽減するために，関数についてはデータ型の定義から自動的に unboxed value を扱うプログラムを生成する方法を提案した [2]．

しかし，このアプローチには，プログラマがデータ構造を変更する手間が大きいことに加え，unbox 化によって引数が先行評価されることになるため，プログラムの意味を変えてしまう危険性がある．

この問題に対して我々は，クロージャ解析 (closure analysis) を応用し，自動的に unboxed value を要素にもつようなデータ表現に転換する方法を考案した．この方法によると，プログラマはプログラムを変更する煩わ

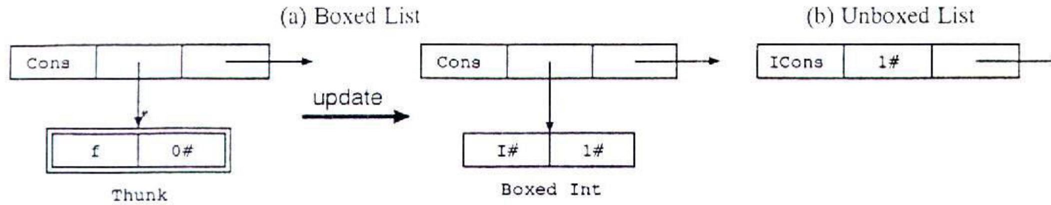

図1 リストの表現と要素の上書き

しさから解放される．

2 ラベル付き言語

対象とする言語を 図 2 に示す．この言語は構成子，case 式，let 式をもつ式で，Peyton-Jones ら [6] の用いている言語と本質的には同一のものである．我々の最適化は，この言語上での変換として実現する．

以下に，この言語の特徴を挙げる：

- ラベル

  クロージャ解析のために，特定の部分式にユニークなラベルがついている．あらかじめ，ラベルなしの言語に対して，構成子・let 式[†1]・ラムダ式に整数を順に割り当て，ラベル付き言語に変換しておく．目的によってはすべての部分式にラベル付けをおこなう方法 [3] もあるが，我々の目的にはすべてにつける必要はない．

  今後，ラベルを ℓ で表し，ラベル ℓ が部分式 e についたものであることを $\ell \Leftarrow e$ で表す．また，適宜，添字を付加することで各ラベルの識別をおこなう：

  - λ ラベル ℓ_λ $(\ell \Leftarrow \lambda^\ell x.e)$
  - 構成子ラベル ℓ_C $(\ell \Leftarrow C^\ell\ x_1 \cdots x_n)$
  - サンクラベル $\ell_{\texttt{let}}$ $(\ell \Leftarrow \texttt{let}\ x = e^\ell)$

- 明示的な box 化

  box 化 を明示的にするため，unboxed value がほぼ第 1 級の対象となっている．unboxed value は，0#,1#, $\cdots$ で表し，整数の boxed value は，構成子 I# を用いて(I# 0#),(I# 1#),$\cdots$ と表す．

- 操作的な意味

  式の評価とヒープへの割当てを明示的に表す．すなわち，case 式でのみ評価をおこない，let 式でのみヒープへの割当てをおこなう．

†1 解析のコストを減らすためには，let 式にラベルをつけるのはサンクの場合だけでよい．図 3 の例ではラベル 4 が省略できる．

$p \in \mathit{Prog}$	::=	binds
$\mathit{bind} \in \mathit{Bind}$	::=	$x{=}e^\ell$
$e \in \mathit{Expr}$	::=	$\mathit{lit} \mid x \mid e\ x \mid \lambda^\ell x.e$
	|	$\oplus\ x_1 \cdots x_n \mid C^\ell\ x_1 \cdots x_n$
	|	case e of alts
	|	let bind in e
$\mathit{alt} \in \mathit{Alt}$	::=	$C\ x_1 \cdots x_n$ -> e | lit -> e
	|	x -> e
$C \in \mathit{Constr}$	::=	Nil | Cons | I#
$\ell \in \mathit{Label}$	::=	$1 \mid 2 \mid \cdots$
$\mathit{lit} \in \mathit{Literal}$	::=	$\cdots$ | 0# | 1# | $\cdots$
$\oplus \in \mathit{Prim}$	::=	+# | -# | /# | $\cdots$
$x \in \mathit{Var}$	:	Variable

図2 ラベル付き言語

以上の特徴に加えて，本論文では次のことを満たしているものとするが，これらの条件を満たすように変換することは容易であり，本質的なものではない：

- 変数名はユニークで，名前の衝突はない．
- 構成子/演算子の引数は常に飽和している (部分適用はない)．
- 関数の引数，構成子の引数は変数である．
- 扱うデータ型は整数 (Int) とリスト型に限定する．

この言語によるプログラムの例を 図 3 に示す．gen n は，$[n, n-1, \cdots, 1]$ というリストを作る関数で，gen' n は，このリストに関数 f を適用したリストを作る関数である．gen は潜在的に unboxed List を生成するようにできるのに対し，gen' では boxed List を生成せざるを得ない．というのは，もし unboxed List にすると，(f a#) が先行評価され，プログラムの意味を変えてし

```
gen = λ¹x#.case x# of
      0# -> Nil²
      y# -> let z = (I#³ x#)⁴ in
            let zs = (case (y# -# 1#) of w# -> gen w#)⁵
            in Cons⁶z zs
gen' = λ⁷a#.case a# of
       0# -> Nil⁸
       b# -> let c = (f a#)⁹ in
             let cs = (case (b# -# 1#) of u# -> gen' u#)¹⁰
             in Cons¹¹c cs
```

図3 プログラム例

まうからである.

3 最適化方法

我々の最適化方法は，クロージャ解析・班分け・表現解析・特殊化の 4 つの処理からなる．以下，これらの処理の概要を 図 3 の例を用いて説明する.

3.1 クロージャ解析

クロージャ解析は大域的なフロー解析の一種で，高階の束縛時解析 (binding time analysis) などに使われる基本的な手法である．本論文では単純な monovariant なクロージャ解析 [4] を拡張し，構成子も扱うようにした.

クロージャ解析は，閉じたプログラム p:

$$p = \{x_1 = e_1; \cdots; x_n = e_n\}$$

に対して，「変数」と「それが束縛される可能性のある，式についたラベル集合」との対応を与える ρ と，「λ についたラベル」と「その λ 式に引数を与えて評価した結果となる可能性のある，式のラベル集合」との対応を与える ϕ との 2 つの環境を次式によって求める:

$$\phi \in \mathit{Label} \to \mathit{LabelSet}$$
$$\rho \in \mathit{Var} \to \mathit{LabelSet}$$
$$\phi(\ell) = \mathcal{P}_e[\![e]\!]\ \phi\rho \qquad \text{for all } \ell \Leftarrow \lambda^\ell x.e$$
$$\rho(y) = \bigcup_{i=1}^{n} \mathcal{P}_v[\![e_i]\!]\ \phi\rho y \quad \text{for all variable } y$$

ここで $\mathcal{P}_e$, $\mathcal{P}_v$ はそれぞれクロージャ解析関数 (図 4)，クロージャ伝搬関数 (図 5) である．$\mathcal{P}_e[\![e]\!]\ \phi\rho$ は，環境 ϕ, ρ のもとで 式 e を評価した結果，どのラベルになるかを求める関数であり，$\mathcal{P}_v[\![e]\!]\ \phi\rho y$ は 式 e を評価する際に変数 y がどのラベル集合に束縛されるかを求める関数である．これらの関数は構成子の拡張部分以外は，[4] と同じである.

$$\mathcal{P}_e[\![\mathit{lit}]\!]\ \phi\rho = \{\}$$
$$\mathcal{P}_e[\![x]\!]\ \phi\rho = \rho(x)$$
$$\mathcal{P}_e[\![\oplus\ x_1 \cdots x_n]\!]\ \phi\rho = \{\}$$
$$\mathcal{P}_e[\![C^\ell x_1 \cdots x_n]\!]\ \phi\rho = \{\ell\}$$
$$\mathcal{P}_e[\![\lambda^\ell x.e]\!]\ \phi\rho = \{\ell\}$$
$$\mathcal{P}_e[\![e\ x]\!]\ \phi\rho = \{\phi(\ell) \mid \ell \in \mathcal{P}_e[\![e]\!]\ \phi\rho\}$$
$$\mathcal{P}_e[\![\texttt{case}\ e\ \texttt{of}\ (c_1; \cdots; c_n)]\!]\ \phi\rho = \bigcup_{i=1}^{n} \mathcal{P}_e[\![e_i]\!]\ \phi\rho$$
$$\text{where } c_i = C_i\ x_{i1} \cdots x_{in_i} \to e_i$$
$$\mathcal{P}_e[\![\texttt{case}\ e\ \texttt{of}\ (l_1; \cdots; l_n)]\!]\ \phi\rho = \bigcup_{i=1}^{n} \mathcal{P}_e[\![e_i]\!]\ \phi\rho$$
$$\text{where } l_i = \mathit{lit}_i \to e_i$$
$$\mathcal{P}_e[\![\texttt{let}\ (x = e^\ell)\ \texttt{in}\ e']\!]\ \phi\rho = \mathcal{P}_e[\![e']\!]\ \phi\rho$$

図 4 クロージャ解析関数

図 3 に対してクロージャ解析をおこなった結果の一部を以下に示す:

$$\rho(\texttt{gen}) = \{1\} \quad \rho(\texttt{z}) = \{3, 4\} \quad \phi(1) = \{2, 6\}$$
$$\rho(\texttt{zs}) = \{5, 2, 6\}$$
$$\rho(\texttt{gen'}) = \{7\} \quad \rho(\texttt{c}) = \{9\} \quad \phi(7) = \{8, 11\}$$
$$\rho(\texttt{cs}) = \{10, 8, 11\}$$

例えば $\phi(1) = \{2, 6\}$ は，gen の生成するリストは，ラ

$$\mathcal{P}_v[\![lit]\!]\ \phi\rho y = \{\} \quad \mathcal{P}_v[\![x]\!]\ \phi\rho y = \{\} \quad \mathcal{P}_v[\![\oplus\ x_1 \cdots x_n]\!]\ \phi\rho y = \{\} \quad \mathcal{P}_v[\![\lambda^\ell x.e]\!]\ \phi\rho y = \mathcal{P}_v[\![e]\!]\ \phi\rho y$$

$$\mathcal{P}_v[\![C^\ell\ x_1 \cdots x_n]\!]\ \phi\rho y = \begin{cases} \rho(x_i) & \text{if } y \text{ is } x_i \\ \{\} & \text{otherwise} \end{cases}$$

$$\mathcal{P}_v[\![e\ x]\!]\ \phi\rho y = \begin{cases} L \cup \rho(x) & \text{if } y \text{ is } x' \text{ where } \ell \in \mathcal{P}_e[\![e]\!]\ \phi\rho, \ell \Leftarrow \lambda^\ell x'.e' \\ L & \text{otherwise} \end{cases} \quad \text{where } L = \mathcal{P}_v[\![e]\!]\ \phi\rho y$$

$$\mathcal{P}_v[\![\texttt{case}\ e\ \texttt{of}\ (c_1;\cdots;c_n)]\!]\ \phi\rho y = \begin{cases} L \cup \rho(x_{ij}) & \text{if } y \text{ is } x_{ij}\ \wedge\ \ell_{C_i} \in \mathcal{P}_e[\![e]\!]\ \phi\rho \\ L & \text{otherwise} \end{cases}$$

$$\text{where } c_i = C_i\ x_{i1} \cdots x_{in_i} \to e_i$$

$$L = \mathcal{P}_v[\![e]\!]\ \phi\rho y \cup \mathcal{P}_v[\![e_1]\!]\ \phi\rho y \cup \cdots \cup \mathcal{P}_v[\![e_n]\!]\ \phi\rho y$$

$$\mathcal{P}_v[\![\texttt{case}\ e\ \texttt{of}\ (l_1;\cdots;l_n)]\!]\ \phi\rho y = L$$

$$\text{where } l_i = lit_i \to e_i$$

$$L = \mathcal{P}_v[\![e]\!]\ \phi\rho y \cup \mathcal{P}_v[\![e_1]\!]\ \phi\rho y \cup \cdots \cup \mathcal{P}_v[\![e_n]\!]\ \phi\rho y$$

$$\mathcal{P}_v[\![\texttt{let}\ (x = e^\ell)\ \texttt{in}\ e']\!]\ \phi\rho y = \begin{cases} L \cup \{\ell\} \cup \mathcal{P}_e[\![e]\!]\ \phi\rho & \text{if } y \text{ is } x \\ L & \text{otherwise} \end{cases} \quad \text{where } L = \mathcal{P}_v[\![e]\!]\ \phi\rho y \cup \mathcal{P}_v[\![e']\!]\ \phi\rho y$$

図 5 クロージャ伝搬関数

ベル 2, 6 の場所で作られる Nil, Cons で構成されることを示している.

3.2 班分け

班分けとは, クロージャ解析によって得られた ρ, ϕ をもとに, 同じ表現をとる必要のある集まり (班) に分割する処理である.

ラベル集合の集合 S :

$$S = \{\phi(\ell) | \ell \in dom(\phi)\} \cup \{\rho(x) | x \in dom(\rho)\}$$

を互いに素な集合 $G_1 \cdots G_N$ に分割する:

$$S = G_1 \cup G_2 \cup \cdots \cup G_N \text{ s.t. } G_i \cap G_j = \{\}$$

分割された各集合 G_i を班と呼ぶ. 分割するには, 共通のラベルを含む集合を同値とみなし, 同値類分解をおこなえばよい. この後, 同じ班に属す構成子の引数の変数は, 同一の班に属すように整理する.

図 3 の場合は, 次の 6 つの班に分割される:

$$G_1 = \{1\}, G_2 = \{5, 2, 6\}, G_3 = \{3, 4\}$$

$$G_4 = \{7\}, G_5 = \{10, 8, 11\}, G_6 = \{9\}$$

3.3 表現解析

班の表現を表現解析によって定める. まず, データの抽象表現 R を次のように定める:

$$R \ ::= \ \texttt{Fun} \mid \texttt{Int} \mid \texttt{Int\#} \mid * \mid R_1 \oplus R_2 \mid \texttt{Nil} \mid \texttt{Cons}\ [R_1, R_2]$$

ここで Fun は関数型の表現, Int/Int# は boxed/unboxed Int の表現である. $R_1 \oplus R_2$ はリストの表現で, R_1, R_2 はそれぞれ Nil , Cons の表現を表す. Cons $[R_1, R_2]$ は, 頭部表現が R_1, 尾部表現が R_2 であるような Cons の表現を表す. Nil は引数をとらないので特殊化できない. * は, 表現の選択に影響を与えないという特別の表現であり, 最終的な表現にはならない.

$G_i = \{\ell_1, \cdots, \ell_N\}$ のとき, ラベルから表現を決める関数δ (図 6) と 演算子 $\sqcup$ (表 1) を使って

$$\Delta(G_i) = \sqcup \delta(\ell_i)$$

によって G_i の表現 $\Delta(G_i)$ を定める. 直観的には, δ によって個々の最適な表現を選び, $\sqcup$ を使って同一の表現に統合している.

δ は, ラベルからその種類に応じて可能な表現の中で最適な表現を返す関数である. ここで重要なのは, $\ell_{\texttt{let}}$ の場合で, 評価コストの安い式 (cheap thunk) の場合は式を先行評価し, unboxed value で表現可能としているところである. これは, 最近のプロセッサでは, サンクの生成に伴うメモリアクセスのコストに比べて, 加算などを投機的に先行評価するコストの方が低いことを考慮したものである. 本論文では cheap thunk を, 1) 評価済

表1 演算子 ⊔

⊔	Fun	Int	Int#	*	Nil	Cons $[r_1, r_2]$
Fun	Fun	⊥	⊥	Fun	⊥	⊥
Int	⊥	Int	Int	Int	⊥	⊥
Int#	⊥	Int	Int#	Int#	⊥	⊥
*	Fun	Int	Int#	*	Nil	Cons $[r_1, r_2]$
Nil	⊥	⊥	⊥	Nil	Nil	Nil ⊕ Cons $[r_1, r_2]$
Cons $[r'_1, r'_2]$	⊥	⊥	⊥	Cons $[r'_1, r'_2]$	Nil ⊕ Cons $[r'_1, r'_2]$	Cons $[r_1 \sqcup r'_1, r_2 \sqcup r'_2]$

$$\delta(\ell_\lambda) = \texttt{Fun} \qquad \delta(\ell_{\texttt{I\#}}) = \texttt{Int\#}$$

$$\delta(\ell_{\texttt{let}}) = \begin{cases} \begin{cases} \texttt{Int\#} & \text{if } e \text{ is a cheap thunk} \\ \texttt{Int} & \text{otherwise} \end{cases} & x \in \texttt{Int} \\ * & \text{otherwise} \end{cases}$$

$$\text{where } \ell \Leftarrow \texttt{let } x = e^\ell$$

$$\delta(\ell_{\texttt{Nil}}) = \texttt{Nil}$$

$$\delta(\ell_{\texttt{Cons}}) = \texttt{Cons } [\Delta(\rho(x_1)), \Delta(\rho(x_2))]$$

$$\text{where } \ell \Leftarrow \texttt{Cons}^\ell\ x_1\ x_2$$

図6 関数 δ

みのもの, 2) 先行評価して問題のない軽い演算のもの[†2], と定めた. 例えば, 次の式は cheap thunk である:

```
1) let z = I# x# in ···
2) let z = case (x# +# y#) of
           z# -> I# z# in ···
```

演算 ⊔ は, 2つの抽象表現のうちでより一般的な表現に統一することを表している. 表1で ⊥ とあるのは定義されないという意味であるが, $x \sqcup y$ で x, y の型が異なる場合はあり得ないことと, 構成子の要素は, 必ず同じ班の表現になることから, $x \sqcup y$ が未定義となるようなケースは生じないことが保証される.

例えば, G_3, G_2 の表現を求める過程は次のようになる:

$$\begin{aligned} \Delta(G_3) &= \delta(3_{\texttt{I\#}}) \sqcup \delta(4_{\texttt{let}}) \\ &= \texttt{Int\#} \sqcup \texttt{Int\#} \\ &= \texttt{Int\#} \end{aligned}$$

†2 例えば, 除算は例外を起こし得るので除外する.

$$\begin{aligned} \Delta(G_2) &= \delta(5_{\texttt{let}}) \sqcup \delta(2_{\texttt{Nil}}) \sqcup \delta(6_{\texttt{Cons}}) \\ &= * \sqcup \texttt{Nil} \sqcup \texttt{Cons } [\Delta(\rho(\texttt{z})), \Delta(\rho(\texttt{zs}))] \\ &= \texttt{Nil} \sqcup \texttt{Cons } [\Delta(G_3), \Delta(G_2)] \\ &= \texttt{Nil} \oplus \texttt{Cons } [\texttt{Int\#}, \Delta(G_2)] \end{aligned}$$

同様に, その他の班の表現は以下のように定められる:

$$\Delta(G_1) = \Delta(G_4) = \texttt{Fun} \qquad \Delta(G_6) = \texttt{Int}$$

$$\Delta(G_5) = \texttt{Nil} \oplus \texttt{Cons } [\texttt{Int}, \Delta(G_5)]$$

3.4 特殊化

表現解析で定められた表現の中で, Int#を要素にもつリスト表現だけに対して特殊化をおこなう. これには相当する代数データ型を新たに作り, 構成子ラベルの構成子を置換すればよい.

図3の例では,

$$\Delta(G_2) = \texttt{Nil} \oplus \texttt{Cons } [\texttt{Int\#}, \Delta(G_2)]$$

に対応する代数データ型:

```
data IList = INil | ICons Int# IList
```

を定義し, Cons, Nil をそれぞれ ICons, INil で置き換え, Cons の第1引数を boxed value (z) から unboxed value (x) に置き換えればよい. 結果として, gen を特殊化した gen_spec という関数が作られる (図7).

4 評価

提案する最適化手法を評価するため, これを Glasgow Haskell compiler [5] に実装した. 実験は, Ultra Sparc 1(Ultra SPARC 167MHz) 上でおこない, 実行時のヒープ/スタックは 20M/8M とした. 最適化をおこなった場合 (Opt) とおこなわない場合 (—) とで, ヒープに割り当てた総量と実行速度を比較した (表2).

```
gen_spec = λx#.case x# of
       0# -> INil
       y# -> let zs = case (y# -# 1#) of
                        w# -> gen_spec w#
             in ICons x# zs
```

図7 特殊化した関数

表2 最適化の効果

Program	Time(sec)		Heap(Mbytes)	
	—	Opt	—	Opt
queen	12.0	6.8	250.9	141.3
wang	7.8	7.8	70.8	70.7

queen は 12 クイーン問題の解の数を数えるプログラムで, 駒の配置を表すために用いられている整数のリストがすべて unbox 化できたために, 大きな効果が現れている.

wang は, Wang のアルゴリズムによる線形方程式解法プログラムであるが, 計算で中心的な役割を果たしている組型の要素が unbox 化できないため, ほとんど効果が現れなかった. 正格性解析と組み合わせると効果が出ると考えられる.

5 関連研究

遅延評価に基づく関数型言語で, 代数データ型を unbox 化 する方法には 2 つのアプローチがある. 1 つは, unbox 化したデータ表現をプログラマが指定する方法である [6] [2]. この方法は, プログラマがデータ構造を変更する手間が大きいことに加え, プログラムの意味を変えてしまう危険性があるという問題がある. もう 1 つは本論文で示した, 大域的な解析による自動的な方法である. 我々が知る限り, このアプローチをとった研究はなされていない. この方法は, プログラムを変更する必要がないので, 第一の方法の問題点を解決しているといえる. 一方で, 解析のコストが無視できないが, クロージャ解析のアルゴリズムの改善によって克服できるものと考えられる.

大域的な解析は種々の最適化に利用されているが, 遅延評価型言語で unbox 化をおこなう目的で利用されている研究は少ない. Faxén [1] は, 型推論規則を利用したフロー解析によって表現選択をするという我々のものと類似した方法を提案しているが, 代数データ型の場合は考慮されてはいない.

6 おわりに

従来の解析手法であるクロージャ解析と表現解析を拡張・変更し, 代数データ型の要素を自動的に unbox 化 する手法を提案した. 本手法に基づき, Glasgow Haskell Compiler のコード生成部を改良し, いくつかのプログラムに対して実行効率の向上を確認した.

我々の提案する最適化方法は自動的におこなわれるため, 大きなプログラムにも効果が期待される. 本論文では単純な monovariant な解析を拡張した方式を示したが, 今後は polyvariant な解析を用いたより精密な解析も考慮する必要があろう.

参考文献

[1] Faxén, K. F. : Optimizing Lazy Functional Programs Using Flow-Inference, *Proc. of the Workshop on Types for Program Analysis*, 1995.

[2] Hall, C., Peyton Jones, S. L. and Sansom, P. : Unboxing using Specialisation, *Proc. of the Glasgow Functional Programming Workshop*, 1994.

[3] Jagannathan, S. and Wright, A. : Flow-directed Inlining, *Proc. of the ACM Conference on Programming Language Design and Implementation*, 1996.

[4] Jones, N. D., Gomard, C. and Sestoft, P. : *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993.

[5] Peyton Jones, S. L., Hall, C., Hammond, K. and Partain, W. : The Glasgow Haskell compiler: a technical overview, *Proc. of the UK Joint Framework for Information Technology Technical Conference*, 1992.

[6] Peyton Jones, S. L. and Launchbury, J. : Unboxed values as first class citizens in a non-strict functional language, *Proc. of the Conference on Functional Programming and Computer Architecture*, 1991.

読書案内 マルチエージェントシステム 大沢英一 横尾真

1 はじめに

これまでに，“マルチエージェントに興味があるんだけど，どんな本を読んだら良いかな?” という質問を受けることが度々あったが，正直なところ何と答えて良いか困惑してしまい，“マルチエージェントの何に興味があるんですか?” と聞き返して，しかるのちに適当な論文等を紹介することでお茶を濁してきた．要は，様々な論文，論文集はあるのだけれど，良い教科書，解説書が見当たらないのである．マルチエージェントシステムと呼ばれる研究分野は，melting pot (るつぼ) と称する人もいるほど，様々な研究分野からなっており，統一的な基礎理論があるわけではない．このような分野の解説は概して，こんな研究もあります，あんな研究もありますという総花的なものになり，読者としては，色々あることは分かるのだけれど，結局，分野の全体像はなんだか良く分からないといったことになりがちである．

しかしながら，近年，独自の観点で良くまとめられた教科書的な本，特集記事がいくつか出されている．本案内ではこれらの紹介を行ない，併せてマルチエージェントシステム関連の論文集等へのポインタを示す．

2 教科書/解説書/解説記事

[9] は，個々のエージェントから出発し，その集団の行動まで，基礎となるモデルを解説した意欲的な本である．最新の個々の研究の総花的な解説ではなく，言語行為，マルチエージェントプランニング，黒板モデル，契約ネットプロトコル，市場モデル等，基本となるようなモデルを詳しく説明している．

[15] はエージェントアーキテクチャ，マルチエージェントシステム，ソフトウェアエージェント，インターフェースエージェントなどに関する研究の意義，経緯，現状，課題，および展望について広く紹介している．

[11] は数十人の研究者による書き下ろしの論文集の形式であるが，分散人工知能の基礎，協調や整合の機構，試験的に開発された分散人工知能システム，関連分野から眺めた分散人工知能，という 4 つの部分から構成されており，それぞれの分野における主要な研究の紹介が網羅的に行なわれている．

[12] は，著者らが分散人工知能，マルチエージェントシステムの研究の初期から一貫して取り組んできた，ゲーム理論に基づく交渉メカニズムに関する研究をまとめた本であり，異なる主体によって設計され，異なる目的を持った複数のプログラムが交渉を行なう場合のルール/プロトコルを設計するための指針を与えている．本書の内容は，オリジナルの文献よりも良く整理されており，例題も豊富で理解しやすいと思われる．

また，[13] は，マルチエージェントシステムを直接扱った章はないが，知的エージェントという観点からまとめられた人工知能の網羅的な教科書であり，共立出版より翻訳の出版が予定されている．

[10] では，エージェントのメンタルモデル，プランニング，学習，交渉といった基礎理論から，ロボットやソ

Ei-Ichi Osawa, (株) ソニーコンピュータサイエンス研究所, Sony Computer Science Laboratory Inc.
Makoto Yokoo, NTT コミュニケーション科学研究所, NTT Communication Science Laboratories.
コンピュータソフトウェア, Vol.14, No.4 (1997), pp.76–77.