26-14-E3

関数型言語処理系におけるデータ構成子の unbox 化
Unboxing Data Constructors in a Lazy Functional Language

神門 有史[†]　田中 哲朗[†]　武市 正人[†]
Yuuji KANDO Tetsurou TANAKA Masato TAKEICHI

†東京大学大学院工学系研究科
Graduate School of Engineering, University of Tokyo

概 要

Haskell などの関数型言語では，リストなどの代数データ型が頻繁に使われる．リストの要素を unboxed value で表現するとヒープ使用量が減少し実行効率も向上することはよく知られているが，遅延評価に基づく言語では要素が必須の場合にのみ可能であるため，これまでは自動的に unbox 化 することは困難を究めていた．本稿ではクロージャ解析という大域的なフロー解析を用いて，自動的に unboxed value を要素としてもつ代数データ型に特殊化する手法を提案し，さらに Glasgow Haskell compiler に実装した上での評価を示す．

1 はじめに

遅延評価型の関数型言語 (Haskell など) は高い表現力をもつが，実行時のコストが高く，先行評価型の言語 (ML, Scheme など) と比べて実行時の性能が劣るという問題点がある．なかでも，遅延評価のコストが高いため，必須性解析 (strictness analysis) をおこない，結局は評価される式を先行評価戦略に変換する方式が不可欠である．

しかし，リストなどの関数型言語で頻繁に使われる，構造をもつデータ (代数データ型) を効率よく表現するのは難しい．例えば，整数のリスト `List Int` はヒープ上では次のような構造で表される (図 1):

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

遅延評価では `Cons` の生成時には，要素に未評価の値

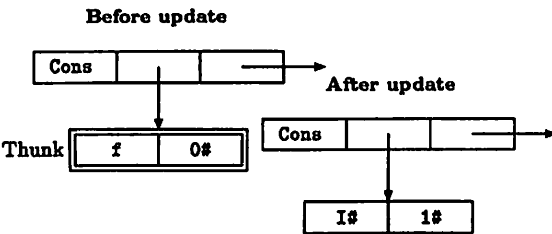

図 1. List Int

図 2. IList

(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．このように構成子の要素としてサンクをもつ必要があるため，ポインタによる間接表現 (boxed value) をとる必要がある．

一方，ポインタを使わない表現 (unboxed value) として，整数自体を格納することができれば，消費されるセル数が減少し効率がよい．それには，多相型`List a`を，`Int#`(unboxed Int の型) について特殊化した `IList` 型:

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

を用いればよい (図 2)． 先行評価型の言語ではコンパイル時に型が分かれば，unboxed value を用いることができる [3]．しかし，遅延評価のもとでは，プログラマが明示的に指定しない限り `IList` のようなデータ表現を使うことができない．この場合，型を特殊化するとそれを引数とする関数も書き換える必要があり，プログラマにとって大きな負担となる．この問題に対して Hall らは，プログラマが，データ型を明示的に unbox 化 するだけで，関数に関しては自動的に特殊化した関数を生成される方法を提案した [1]．しかしこの方法には，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，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.

そこで, 我々はクロージャ解析 (closure analysis) によって自動的に unboxed value を要素にもつようなデータ表現に転換する方法を提案する. この方法によって, プログラマはプログラムを変更する煩わしさから解放されることになる.

2 ラベルつき言語

対象とする言語を 図 3 に示す. この言語は構成子 /case 式 /let 式をもつラムダ式であり, コンパイラの中間言語として使われる. 後述する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.

簡単のため, この言語は構文規則で示されること以外にも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るので本質的ではない:

- 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.
- let 式 の *bind* は一つとする.
- 扱うデータ型は整数 (Int) とリスト型に限定する.

この言語では, プログラムは *bind* の集まりで, その中に main 関数があり, その値をプログラムの結果とする. 0#,1#, ⋯ は, unboxed value の定数で, I# 1# のように boxed Int を構成子 (I#) で表す.

この言語は, 必須性解析の結果をもとにプログラム変換された後のものであるとする. 従って, 式の評価とヒープへの割り当てが明示的に表される. つまり, case 式でのみ評価をおこない, let 式でヒープへの割り当てをおこなう. 例えば以下のプログラムの断片では, z には, unboxed value x# を要素とする boxed Int をヒープ上に作り, それを指すポインタが入る.

```
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
```

ここで, gen n は, $[n, n-1, \cdots, 1]$ というリストを作る関数で, gen' n は, さらに関数 f を適用したリストを作る関数である.

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

図 3. ラベル付き言語

3 最適化方法の概要

提案する最適化方法を説明するため, 前節のプログラムを最適化する手順を示す:

(1) クロージャ解析

「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」との対応与える ρ と, 「λ についたラベル」と「その λ 式の本体が評価されてなる可能性のある, 式のラベル集合」との対応を与える ϕ を得る:

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

(2) 班分け

同じ変数を束縛するラベル集合は統合され, 次の 6 つの班に分かれる:

$$G_1 = \{1\}, G_2 = \{5,2,6\}, G_3 = \{3,4\}$$
$$G_4 = \{7\}, G_5 = \{10,8,11\}, G_6 = \{9\}$$

(3) 表現解析

表現解析をおこない, 各班の表現を求める:

$\Delta(G_1) = \Delta(G_4) = \text{Box}$
$\Delta(G_2) = \text{Cons [Int\#}, \Delta(G_2)]$
$\Delta(G_3) = \text{Int\#}, \Delta(G_6) = \text{Int}$
$\Delta(G_5) = \text{Cons [Int}, \Delta(G_5)]$

(4) 特殊化

$\Delta(G_2)$ はリストの要素の型が Int# にできることを示唆しており, gen の Nil,Cons を, それぞれ INil,ICons に置き換えることで, unboxed value をもつ特殊化したリストを用いることができる. $\Delta(G_5)$ には Int が残っているので, gen' の作るリストは特殊化することができない.

4 必須性解析と投機的先行評価

遅延評価を実現するためにサンク構造が用いられる. サンクは, 包み込む式の自由変数を要素としてもつようにヒープ上に表現される. サンクを用いるために次の 2 種

類のコストがかかる:

- Allocation : ヒープへの書き込みのコスト
- Update : 評価後の値で書き換えるコスト

このため遅延評価型の関数型言語では，必須性解析をおこない，先行評価が可能な式を特定することによって，可能な限りサンクの生成を防ぐ方法の研究がおこなわれている．ところが，必須でない式であっても安全に評価でき，評価のコストが安いものならば投機的に先行評価する方が得なことが多い．例えば，次のような場合:

```
let v = case (x# +# 1#) of w# -> I# w#
```

は，サンクを作るよりも評価をおこなった方が有利である．我々は，サンクを評価すべきかどうか判断するコスト関数を適当に定め，上記の例のような場合は先行評価をおこなうようにした．

5 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，主に高階の束縛時解析 (binding time analysis) などのために使われる．本稿では単純な monovariant なクロージャ解析 [2]を拡張し，構成子も扱うようにした．

5.1 ラベル付け

ラベル付けは，部分式にユニークなラベル (整数) を与える処理で，ラベルはプログラムテキスト上の出現を特定するために用いられる．最適化の目的によっては，すべての部分式にラベルを付ける方法もある [4]が，我々の目的には，構成子，let 式，ラムダ式にラベルを付けるだけで十分である．実際には，解析のコストは減らすために，let 式にラベルを付けるのはサンクの場合だけでよい．

今後，ラベル一般を表すのに ℓ を用い，ラベル ℓ が部分式 e に付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 — $\ell_{\texttt{let}}$ $(\ell \Leftarrow \texttt{let}\ x = e^\ell)$

5.2 クロージャ解析

クロージャ解析は形式的には以下の 2 つの環境を求める解析である:

$$\phi \in ResEnv = Label \rightarrow LabelSet$$
$$\rho \in VarEnv = Var \rightarrow LabelSet$$
$$\begin{cases} \phi(\ell) & \text{— } \ell_\lambda \text{ の本体の評価結果のラベル集合} \\ \rho(x) & \text{— 変数 } x \text{ に束縛されるラベル集合} \end{cases}$$

クロージャ解析関数 $\mathcal{P}_e$ (図 4) とクロージャ伝搬関数 $\mathcal{P}_v$ (図 5) を用いて，ϕ, ρ を求める (実装では分割コンパイルを考慮しているが本稿では省略する)．$\mathcal{P}_e[\![e]\!]\ \phi\rho$ 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

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

図 4. クロージャ解析関数

$$\mathcal{P}_v[\![c]\!]\ \phi\rho y = \{\} \quad \mathcal{P}_v[\![x]\!]\ \phi\rho y = \{\} \quad \mathcal{P}_v[\![\oplus\ x_1 \cdots x_n]\!]\ \phi\rho y = \{\}$$
$$\mathcal{P}_v[\![C^\ell\ x_1 \cdots x_n]\!]\ \phi\rho y = \begin{cases} \rho(x_i) & \text{if } y \text{ is } x_i \\ \{\} & \text{otherwise} \end{cases}$$
$$\mathcal{P}_v[\![\lambda^\ell x.e]\!]\ \phi\rho y = \mathcal{P}_v[\![e]\!]\ \phi\rho y$$
$$\mathcal{P}_v[\![e\ x]\!]\ \phi\rho y$$
$$= \begin{cases} L \cup \rho(x) & \text{if } y \text{ is } x' \text{ where } \ell \in \mathcal{P}_e[\![e]\!]\ \phi\rho, \ell \Leftarrow \lambda^\ell x'.e' \\ L & \text{otherwise} \end{cases}$$
$$\quad \text{where } L = \mathcal{P}_v[\![e]\!]\ \phi\rho y$$
$$\mathcal{P}_v[\![\texttt{case}\ e\ \texttt{of}\ (c_1; \cdots; c_n)]\!]\ \phi\rho y$$
$$= \begin{cases} L \cup \rho(x_{ij}) & \text{if } y \text{ is } x_{ij}\ \wedge\ \ell_{C_i} \in \mathcal{P}_e[\![e]\!]\ \phi\rho \\ L & \text{otherwise} \end{cases}$$
$$\quad \text{where } c_i = C_i\ x_{i1} \cdots x_{in_i} \rightarrow e_i$$
$$\qquad L = \mathcal{P}_v[\![e]\!]\ \phi\rho y \cup \mathcal{P}_v[\![e_1]\!]\ \phi\rho y \cup \cdots \cup \mathcal{P}_v[\![e_n]\!]\ \phi\rho y$$
$$\mathcal{P}_v[\![\texttt{case}\ e\ \texttt{of}\ (l_1; \cdots; l_n)]\!]\ \phi\rho y = L$$
$$\quad \text{where } l_i = lit_i \rightarrow e_i$$
$$\qquad L = \mathcal{P}_v[\![e]\!]\ \phi\rho y \cup \mathcal{P}_v[\![e_1]\!]\ \phi\rho y \cup \cdots \cup \mathcal{P}_v[\![e_n]\!]\ \phi\rho y$$
$$\mathcal{P}_v[\![\texttt{let}\ (x = e^\ell)\ \texttt{in}\ e']\!]\ \phi\rho y = \begin{cases} L \cup \{\ell\} \cup \mathcal{P}_e[\![e]\!]\ \phi\rho & \text{if } y \text{ is } x \\ L & \text{otherwise} \end{cases}$$
$$\quad \text{where } L = \mathcal{P}_v[\![e]\!]\ \phi\rho y \cup \mathcal{P}_v[\![e']\!]\ \phi\rho y$$

図 5. クロージャ伝搬関数

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．ϕ, ρ は，プログラム $p = \{x_1 = e_1; \cdots; x_n = e_n\}$ に対して次式を満たすように定める:

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

6 代数データ型の特殊化

クロージャ解析によって，データが作られた場所がラベル集合として近似的に求められる．同じ場所で使われるデータは同一の表現をとる必要がある．逆に，作られる場所と使われる場所が異なれば，別の表現を選ぶことができ，unboxed value を要素にもつ特殊化したデータ構成子を使う可能性が生まれる．我々は，班分けによってそれぞれ独立して表現を選べる集まり (班) を求め，表現解析によって各班の表現を決定する方法を提案する．この方

法は Faxén の表現解析 [5]を構成子に関して拡張したものとなっている.

6.1 班分け

ラベル集合の集合 S:

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

を班に分ける. 同じラベル集合に属しているラベルを同値とみなし, 同値類分解をおこなって:

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

という互いに素な班に分割できる. 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する.

6.2 表現解析

各班 G_i の表現 $\Delta(G_i)$ を表現解析によって決定する. データの表現 R を以下のように定める:

R ::= `Box` | `Int` | `Int#` | `Cons` $[R_1, R_2]$ | $*$ | $\Delta(G_i)$

ここで `Box` はすべての boxed value , `Int#` は unboxed Int の表現である. `Cons` $[R_1, R_2]$ は頭部表現が R_1, 尾部表現が R_2 であるようなリスト表現を表す. `Nil` は引数をとらないのでここでは考えない. $\Delta(G_i)$ は班 G_i の表現を表し, 再帰的な型の表現を表す場合に使う. $*$ 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, 以下で定義する δ と $\sqcup$ を使って $\Delta(G_i) = \sqcup\delta(\ell_i)$ によって G_i の表現を定めることができる.

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

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

$$\delta(\ell_{\lambda}) = \texttt{Box} \qquad \delta(\ell_{\texttt{I\#}}) = \texttt{Int\#} \qquad \delta(\ell_{\texttt{Nil}}) = *$$

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

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

ここで $\ell_{\texttt{let}}$ で安いサンクの場合は `Int#` としている. また grp はラベル集合の班を返す関数である.

$\sqcup$ は次のように定める:

$$\texttt{Int} \sqcup \texttt{Int\#} = \texttt{Int} \quad x \sqcup * = x \quad x \sqcup x = x$$

ここで $x \sqcup y = y \sqcup x$ とする.

6.3 特殊化

表現解析で定まった表現の中で, unboxed value を要素にもつような表現だけに注目して特殊化をおこなう. これは相当する構成子を作り, 構成子ラベルの構成子を置き換えることで実現できる. ただし, 外部で使われる可能性のある関数の場合, その引数の型を特殊化することはできない.

表 1. 最適化の効果

Program	Time(sec)		Heap(Mbytes)	
	—	Opt	—	Opt
kkqueen	12.0	6.8	250.9	141.3
wang	2.0	2.0	27.7	27.4

7 評 価

提案する最適化手法を評価するため, Glasgow Haskell compiler 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量と実行速度を比較した (表 1). どちらも `-O2 -fglasgow-exts` オプションを付けてコンパイルした. またコスト関数は加算・減算・乗算などからなる式を安いとするものを用いた.

kkqueen は 12 クイーン問題を解くプログラムで, 結果の表すのに使われている整数のリストがすべて特殊化できたために, 大きな効果があらわれている. その一方で wang では, 組型の要素が unbox 化できないため, ほとんど変わらない. 必須性解析と組み合わせると効率がよくなると思われる.

8 おわりに

我々の提案する最適化方法は全て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. 本稿では単純な monovariant な解析を拡張した方式を示したが, クロージャ解析においてもアルゴリズムの改良による高速化も重要である. 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] Hall, C. and Jones, SL, P. and Sansom, P. : Unboxing using Specialisation, *Proc. of the Glasgow functional programming workshop*, 1994.

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

[3] Leroy, X. : Unboxed objects and polymorphic typing, POPL'92, 1992.

[4] Jagannathan, S. and Wright, A. : Flow-directed Inlining, PLDI'96, 1996.

[5] Faxén, KF. : Optimizing Lazy Functional Programs Using Flow-Inference, *Proc. of the workshop on Types for Program Analysis*, 1995.