日本ソフトウェア科学会第 28 回大会 (2011 年度) 講演論文集

低水準コード生成を行う $\lambda^{\bigcirc}$ 仮想機械の融合変換を使った系統的導出

小山内 幸一　五十嵐 淳

本論文では多段階計算を表現する計算体系 $\lambda^{\bigcirc}$ のための仮想機械の系統的導出について議論する．

多段階計算とは，部分計算などコード生成を行うシステムをモデル化した計算体系であり，$\lambda^{\bigcirc}$ や $\lambda^{\bigcirc\Box}$ など様々な体系が提案されている．このうち，$\lambda^{\bigcirc}$ は擬似引用機構を備えた λ 計算であり，引用された $\lambda^{\bigcirc}$ 項をコード生成の結果とすることができる．

Igarashi と Iwaki は，Ager らの手法を応用し，$\lambda^{\bigcirc}$ のインタプリタから，プログラム変換により，コンパイラと仮想機械プログラムを導出した．ここで導出された仮想機械で生成されるコードは $\lambda^{\bigcirc}$ の意味論を反映して $\lambda^{\bigcirc}$ の項になっている．さらに，彼らは低水準コード即ち仮想機械命令列の生成を直接行うような仮想機械も示したが，二つの仮想機械の関係は明らかではない．

本研究では，前者の仮想機械とコンパイラ・逆コンパイラの関数合成が後者になることに着目し，両者の関係を融合変換を使って定式化する．

1　はじめに

$\lambda^{\bigcirc}$ [3] は擬似引用機構を備えた λ 計算である．$\lambda^{\bigcirc}$ は多段階計算を表現していて，部分計算などコード生成を行うシステムをモデル化する．この体系ではコード生成の結果として $\lambda^{\bigcirc}$ 項を生成することができる．

本論文では $\lambda^{\bigcirc}$ の処理系である仮想機械の系統的導出について議論する．Igarashi と Iwaki [4] は，Ager らの手法 [1] を応用し，$\lambda^{\bigcirc}$ のインタプリタから，プログラム変換により，コンパイラと仮想機械プログラムを導出した．ここで導出された仮想機械で生成されるコードは $\lambda^{\bigcirc}$ の意味論を反映して $\lambda^{\bigcirc}$ の項になっている．そのため生成されたコードを実行するには生成された項をコンパイルしてから仮想機械に入力する必要がある．

彼らは低水準コード即ち仮想機械命令列の生成を行うような仮想機械も示した．この仮想機械を使うと，次の段階の計算に進むときに生成された命令列をそのまま仮想機械に入力することができる．しかし，このふたつの仮想機械の間の関係は明らかではなかった．

本論文ではこの低水準コードの生成を行う仮想機械の系統的な導出方法を示す．高水準コード生成を行う仮想機械とコンパイラ・逆コンパイラの合成関数が低水準コードの生成を行う仮想機械となることに着目している．また，融合変換という方法を用いてこの合成関数を変形することで，Igarashi と Iwaki が示した低水準コード生成を行う仮想機械を導出する．

本論文の構成は以下のようになっている．2 節，3 節では，それぞれ，$\lambda^{\bigcirc}$ と Igarashi と Iwaki による高水準コード生成を行う仮想機械について述べる．その後 4 節で低水準コードの生成を行う仮想機械を系統的に導出する．5 節では関連研究や今後の課題について述べる．

Systematic Derivation of a $\lambda^{\bigcirc}$ Virtual Machine with Low-Level Code Generation by using Fusion Transformation

Koichi Osanai and Atsushi Igarashi, 京都大学大学院情報学研究科知能情報学専攻, Department of Intelligence Science and Technology, Graduate School of Informatics, Kyoto University.

2 計算体系 $\lambda^{\bigcirc}$

$\lambda^{\bigcirc}$ とは多段階計算を表現する λ 計算の拡張の一つである．多段階計算とは，部分計算などコード生成を行うシステムをモデル化した計算体系である．$\lambda^{\bigcirc}$ は擬似引用機構を備えており，$\lambda^{\bigcirc}$ 項をコード生成の結果とすることができる．$\lambda^{\bigcirc}$ は線形時間時相論理に対応する型システムを持っているが [3]，本研究の範囲では型は関係ないので，以下でも型については議論しない．

$\lambda^{\bigcirc}$ では，擬似引用 (Lisp や Scheme での `quasiquote` または ‘) を **next** で表し，引用からの脱出 (Lisp や Scheme での `unquote` または `,`) を **prev** で表す．例えば $(\lambda x.\,\mathbf{next}((\lambda y.y)\,\mathbf{prev}\,x))(\mathbf{next}\,\lambda z.z)$ という項を考える．この項の前半は，(コードを動く) 引数 x に $\lambda y.y$ を適用するコードを返す関数であり，それを恒等関数のコード $\mathbf{next}\,\lambda z.z$ に適用している．この項を簡約すると，**prev** と **next** が打ち消しあって，$\mathbf{next}(\lambda y.y)(\lambda z.z)$ という引用項が得られる．

2.1 構文

$\lambda^{\bigcirc}$ の項 t は以下の構文で与えられる．

$$t \quad ::= \quad n \mid \lambda t \mid t_0\,t_1 \mid \mathbf{next}\,t \mid \mathbf{prev}\,t$$

ここで n は自然数であり，de Bruijn index を使って，束縛変数を表現する．変数 n は同じ引用レベルでの n 番目の束縛変数を指す．ある部分項の引用レベルとは，その部分項の外側にある (**next** の数) − (**prev** の数) である．例えば，$\lambda y.\,\mathbf{next}(\lambda x.x(\mathbf{prev}\,y))$ は $\lambda\,\mathbf{next}(\lambda 0(\mathbf{prev}\,0))$ と表わされる (x はレベル 1 で束縛されているが，y はレベル 0 で束縛されていることに注意．)

2.2 意味論

$\lambda^{\bigcirc}$ の意味論は，レベル 0 の部分項のみを通常通り計算し，レベル 1 以上の部分項は引用の内部なのでそのまま残すようなものになっている．評価の結果である値は関数閉包か項の引用である．

値 v と環境 E を以下のように定義する．

$$v \quad ::= \quad \langle E, t\rangle \mid \ulcorner t \urcorner$$
$$E \quad ::= \quad \cdot \mid v :: E$$

$\langle E, t\rangle$ が関数閉包，$\ulcorner t \urcorner$ が項の引用を表している．

評価関係 $E \vdash t \Downarrow^{l} v$ は，レベル l の項 t を環境 E で評価すると v になることを表す．項の評価規則は図 1 のようになっている．ここで，$l > 1$ である．

レベル 0 の変数，関数抽象，関数適用の評価は通常の (値呼び)λ 計算の通りである．レベル 1 以上の変数，関数抽象，関数適用の評価は **prev** によってレベル 0 になっている部分項のみを評価して，他はそのまま残す．

規則 E-Var に現れる $E(n)$ は環境 E の n 番目の値を表す．規則 Eq-Abs 中に現れる $E \uparrow^{l}$ はレベル l の変数のインデックスを+1 するシフト演算を表している．例えば，$E = \ulcorner 0 \urcorner :: \ulcorner \mathbf{next}\,0 \urcorner :: \cdot$ の時，$E \uparrow^{1} = \ulcorner 1 \urcorner :: \ulcorner \mathbf{next}\,0 \urcorner :: \cdot$ となる．

3 $\lambda^{\bigcirc}$ コンパイラと高水準コード生成を行う仮想機械

Igarashi と Iwaki は，Ager らの手法 [1] を応用し，$\lambda^{\bigcirc}$ のインタプリタから，プログラム変換により，コンパイラと仮想機械プログラムを導出した [4]．本研究ではこの仮想機械に対して更なる変換を行うことで低水準コード生成を行う仮想機械を導出する．本節では Igarashi と Iwaki が導出したコンパイラと仮想機械を紹介する．以降では，コンパイラや仮想機械のプログラムは OCaml を使って示す．

まず，$\lambda^{\bigcirc}$ 項を OCaml のデータ構造で表すと図 2 のようになる．

コンパイラの生成する命令は図 3 の型 `inst` で定義されている．各命令の基本的な動作は次のようになる．

- `Access`: 環境から値を取り出す
- `Close`: 関数閉包を作る
- `Push`: 引数の評価を継続に追加する
- `Enter`: レベルを 1 上げる
- `Leave`: レベルを 1 下げる

一方，レベル 1 以上 (`Leave` はレベル 2 以上) のときには，各命令は対応した項の生成を行う．

図 4 の関数 `compile` はコンパイラを表している．このコンパイラの変換は項のそれぞれの構成子が一命令に対応していて，変換前後で同じ構造を持ってい

$$\frac{(E(n) = v)}{E \vdash n \Downarrow^0 v} \text{(E-Var)} \qquad \frac{}{E \vdash n \Downarrow^l n} \text{(Eq-Var)}$$

$$\frac{}{E \vdash \lambda t \Downarrow^0 \langle E, t\rangle} \text{(E-Abs)} \qquad \frac{E \uparrow^l \vdash t \Downarrow^l t'}{E \vdash \lambda t \Downarrow^l \lambda t'} \text{(Eq-Abs)}$$

$$\frac{E \vdash t_0 \Downarrow^0 \langle E', t\rangle \quad E \vdash t_1 \Downarrow^0 v \quad v :: E' \vdash t \Downarrow^0 v'}{E \vdash t_0 t_1 \Downarrow^0 v'} \text{(E-App)} \qquad \frac{E \vdash t_0 \Downarrow^l t_0' \quad E \vdash t_1 \Downarrow^l t_1'}{E \vdash t_0 t_1 \Downarrow^l t_0' t_1'} \text{(Eq-App)}$$

$$\frac{E \vdash t \Downarrow^1 t'}{E \vdash \mathbf{next}\, t \Downarrow^0 \ulcorner t' \urcorner} \text{(E-Next)} \qquad \frac{E \vdash t \Downarrow^{l+1} t'}{E \vdash \mathbf{next}\, t \Downarrow^l \mathbf{next}\, t'} \text{(Eq-Next)}$$

$$\frac{E \vdash t \Downarrow^0 \ulcorner t' \urcorner}{E \vdash \mathbf{prev}\, t \Downarrow^1 t'} \text{(E-Prev)} \qquad \frac{E \vdash t \Downarrow^{l+1} t'}{E \vdash \mathbf{prev}\, t \Downarrow^l \mathbf{next}\, t'} \text{(Eq-Prev)}$$

図 1 $\lambda^{\bigcirc}$ の評価規則

```
type term = Var of int | Abs of term | App of term * term | Next of term | Prev of term
```

図 2 $\lambda^{\bigcirc}$ 項の定義

```
type inst = Access of int | Close of inst list | Push of inst list | Enter | Leave
```

図 3 命令セット

```
(* compile : term -> inst list *)
let rec compile t = match t with
    Var n -> [Access n]
  | Abs t0 -> let c0 = compile t0 in [Close c0]
  | App (t0, t1) -> let c0 = compile t0 and c1 = compile t1 in Push c1::c0
  | Next t0 -> let c0 = compile t0 in Enter::c0
  | Prev t0 -> let c0 = compile t0 in Leave::c0
```

図 4 `compile` : コンパイラ

る．関連研究 [6] では，ひとつの項構成子をより基本的な命令の列へ変換するコンパイラを導出する手法が提案されているが，それに対しても本研究の手法は適用できると考えられる．

以下では，高水準コード生成を行う仮想機械に関する関数やデータ構造の定義は `module H` で行う．

図 5 は仮想機械で扱うデータ構造を表している．型 `value` は項を評価した結果の値を表していて，関数閉包または引用項である．型 `env` は環境で，値のリストである．型 `cont` は継続で，仮想機械の状態の一部である．

図 6 は仮想機械を表している．関数 `vm` は型 `vmstate` 即ち命令列，レベル，環境，継続の組をとり，値を返す．関数 `vmK` は型 `vmKstate` 即ち継続と値の組をとり，値を返す．これらは相互再帰の形になっていて，`vmK` に `Halt` が入力されたときに最終的に停止する．

4 低水準コード生成を行う仮想機械の導出

3 節で示した仮想機械は $\lambda^{\bigcirc}$ の意味論を反映して $\lambda^{\bigcirc}$ の項を生成する．そのため，生成されたコードの実行を行うには生成された項をコンパイルしてから仮想機械に入力する必要がある．本節では $\lambda^{\bigcirc}$ の項の代わりに低水準コード即ち仮想機械命令列を直接生成するような仮想機械を導出する．この仮想機械を

```
module H
  type value = Clos of env * inst list | Q of term
  and env = value list
  and cont = Halt | EvArg of inst list * env * cont | EvBody of value * cont
           | Quote of cont | Unquote of cont | QAbs of cont
           | QApp' of inst list * int * env * cont | QApp of term * cont
           | QNext of cont | QPrev of cont
  type vmstate = inst list * int * env * cont
  type vmKstate = cont * value
```

図 5　高水準コード生成を行う仮想機械の扱うデータ構造

使うと，次の段階の計算に進むときに生成された命令列をそのまま仮想機械に入力するだけでよい．

低水準コード生成を行う仮想機械の仕様を形式的に書くと図 7 のようになる．ここで，L は低水準コード生成を行う仮想機械に関する関数やデータ構造の定義を含むモジュールであり，L.vm が導出すべき仮想機械である．また，comp_env は環境中に現れるコードを項から命令列に変換する関数であり，4.1.3 節で詳しく述べる．

4.1　低水準コード生成を行う仮想機械の表現

4.1.1　データ構造の低水準化

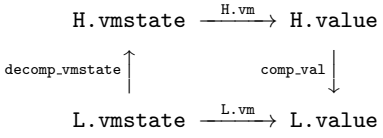
低水準仮想機械を表現するためには，扱うデータ構造を変更する必要がある．変更されたデータ構造の定義を図 8 に示す．まず，値に含まれる項の引用を命令列の引用に変更する必要がある．このようにすることで図 7 の下線部のように命令列の引用が可能になる．また，それに伴ない環境，継続の中に出現するコードも命令列になる．

4.1.2　仮想機械の関数合成を用いた表現

3 節で示した仮想機械 H.vm は，H.vmstate -> H.value という型の関数だった．これに対し低水準コード生成を行う仮想機械 L.vm の型は L.vmstate -> L.value となるはずである．L.vm は，データ構造を変換する関数を用いて以下の三つの関数の合成で表現できる．

1. データ構造を高水準化する関数 decomp_vmstate: L.vmstate -> H.vmstate
2. 高水準コード生成を行う仮想機械 H.vm: H.vmstate -> H.value
3. 値を低水準化する関数 comp_val: H.value -> L.value

つまり，入力とした低水準データを逆コンパイラで高水準化し，それを高水準コード生成を行う仮想機械で処理し，その結果生成された項をコンパイルすれば図 7 の仕様を満たし低水準コード生成を行う仮想機械になるはずである．これを可換図式で表すと以下のようになる．



4.1.3　高水準データ構造との対応

ここで，図 8 で定義されたデータ構造と図 5 のデータ構造との対応を議論する．

値，環境，継続の変換をする関数 comp_val, comp_env, comp_cont を考える．H.value を L.value に変換するには，図 9 のようにすればよい． 項の引用だった部分は，その項をコンパイルした結果の命令列の引用となっている．また，環境や継続に対しても，その中に現れる値をコンパイルすることで H.env から L.env へ, H.cont から L.cont へ変換できる．

同様に，逆方向への変換 (逆コンパイラ) も図 10 のように考えることができる．

```
module H
  (* vm : vmstate -> value *)
  let rec vm = function
      ([Access n], 0, e, k) -> vmK (k, List.nth e n)
    | ([Access n], lv, e, k) -> vmK (k, Q (Var n))
    | ([Close c0], 0, e, k) -> vmK (k, Clos (e, c0))
    | ([Close c0], lv, e, k) -> vm (c0, lv, shiftE(e,lv), QAbs k)
    | (Push c1::c0, 0, e, k) -> vm (c0, 0, e, EvArg (c1, e, k))
    | (Push c1::c0, lv, e, k) -> vm (c0, lv, e, QApp' (c1, lv, e, k))
    | (Enter::c0, 0, e, k) -> vm (c0, 1, e, Quote k)
    | (Enter::c0, lv, e, k) -> vm (c0, lv + 1, e, QNext k)
    | (Leave::c0, 1, e, k) -> vm (c0, 0, e, Unquote k)
    | (Leave::c0, lv, e, k) -> vm (c0, lv - 1, e, QPrev k)
  (* vmK : vmKstate -> value *)
  and vmK = function
      (Halt, v) -> v
    | (EvArg (c1, e, k), v) -> vm (c1, 0, e, EvBody (v, k))
    | (EvBody (Clos (e', c'), k), v) -> vm (c', 0, v::e', k)
    | (Quote k, Q v) -> vmK (k, Q v)
    | (Unquote k, Q v) -> vmK (k, Q v)
    | (QAbs k, Q v) -> vmK (k, Q (Abs v))
    | (QApp' (c1, lv, e, k), Q v0) -> vm (c1, lv, e, QApp (v0, k))
    | (QApp (v0, k), Q v1) -> vmK (k, Q (App (v0, v1)))
    | (QNext k, Q v) -> vmK (k, Q (Next v))
    | (QPrev k, Q v) -> vmK (k, Q (Prev v))
```

図 6 vm, vmK：**仮想機械**

```
H.vm (compile t, 0, [], H.Halt) = H.Q t' ⟹
L.vm (compile t, 0, [], L.Halt) = L.Q (compile t')
かつ
H.vm (compile t, 0, [], H.Halt) = H.Clos (e, c) ⟹
L.vm (compile t, 0, [], L.Halt) = L.Clos (comp_env e, c)
```

図 7 低水準コード生成を行う仮想機械の仕様

ここで decomp は任意の項 t に対して

```
decomp (compiler t) = t
```

となるような関数である．環境と継続に関しても同様に逆コンパイラが定義できる．

4.2 再帰関数の融合変換

二つの再帰関数の合成関数を，一つの関数に置き替える手法 (融合変換) が，Ohori と Sasano によって提案されている [5]．本研究ではこの手法を使って 4.1.2 節で示された 3 つの関数の融合変換を行う．本節では

```
module L
  type value = Clos of env * inst list | Q of inst list
  and env = value list
  and cont = Halt | EvArg of inst list * env * cont | EvBody of value * cont
           | Quote of cont | Unquote of cont | QAbs of cont
           | QApp' of inst list * int * env * cont | QApp of term * cont
           | QNext of cont | QPrev of cont
  type vmstate = inst list * int * env * cont
  type vmKstate = cont * value
```

図 8　低水準コード生成を行う仮想機械の扱うデータ構造

```
let rec comp_val v =
  match v with
      H.Clos (e, c) -> L.Clos (comp_env e, c)
    | H.Q t -> L.Q (compile t)
```

図 9　H.value から L.value への変換

```
let rec decomp_val v =
  match v with
      L.Clos (e, c) -> H.Clos (decomp_env e, c)
    | L.Q is -> H.Q (decomp is)
```

図 10　L.value から H.value への変換

この手法の紹介をする.

例として以下のような関数 mapsq と sum を考える.

```
let rec mapsq l = match l with
    [] -> []
  | h :: t -> (h * h) :: (mapsq t)
let rec sum l = match l with
    [] -> 0
  | h :: t -> h + sum t
```

mapsq はリストの各要素を二乗する関数で，sum はリストの各要素の和を計算する関数である．これら二つの合成関数は，次のように変形することができる.

1. `let sum_mapsq l = sum (mapsq l)`
   (mapsq のインライン展開)
2. `let sum_mapsq l = sum (match l with`
   `  [] -> []`
   `| h :: t -> (h * h) :: (mapsq t))`
   (sum の match による分岐への分配)
3. `let sum_mapsq l = match l with`
   `  [] -> sum []`
   `| h :: t -> sum ((h * h) :: (mapsq t))`
   (sum のインライン展開)
4. `let sum_mapsq l = match l with`
   `  [] -> 0`
   `| h :: t -> h * h + sum (mapsq t)`
   (sum ∘ mapsq を再帰呼び出しで置き換え)
5. `let rec sum_mapsq l = match l with`
   `  [] -> 0`
   `| h :: t -> h * h + sum_mapsq t`

このような変形で，二つの再帰関数の合成関数を一つの再帰関数にすることができる．これによって，中間データである各要素を二乗したリストが生成されることがなくなる.

4.3 高水準コード生成を行う仮想機械とコンパイラ・逆コンパイラの融合変換

本節では，前節で紹介した手法で 4.1.2 節で示された 3 つの関数の融合変換を行う．図 6 で高水準コード生成を行う仮想機械を，図 9 で H.value から L.value への変換をそれぞれ示しているので，残りの逆コンパイラを使ったデータ構造変換の関数を図 11 に示す．この関数は match 式で定義されているが，処理内容は match での分岐とは無関係で，四つ組の中の環境と継続を高水準のものに変換している．match 式で分岐しているのは融合変換がうまくいくようにするためである．この match の分岐は図 6 の H.vm の分岐のパターンに対応している．

これら三つの関数を融合変換する．まず，H.vm と comp_val を融合変換すると，図 12 の comp_Hvm のようになる． match 節の中に comp_val を分配して，その後 comp_val ∘ H.vm を再帰呼び出しで置き換えている．H.vmK に対しても同様に融合変換をする必要がある．

次に decomp_vmstate と comp_Hvm を合成すると図 13 の L.vm のようになる． この関数が導かれる過程は以下のようになる．まず，decomp_vmstate の各 match 節の分岐に comp_Hvm を分配する．例えば，図 11 の枠線で囲まれた部分は

```
comp_Hvm (Enter :: c0, 0,
          decomp_env e, decomp_cont k)
```

のようになる．この部分は，さらに図 14 のように変換されていく． ここで，最後の変形で comp_env (decomp_env e) が e に変形できるのは，ある e' に対して e = comp_env e' の形をしていて，

```
  (comp_env ∘ decomp_env ∘ comp_env) e'
= comp_env e'
= e
```

となるためである．下線部は decomp の定義により恒等関数となる．

もう一つの変換の例として，図 6 の下線部を含む節がどう変換されるかを示す．さきほどと同様に考えると，comp_Hvm を分配した後の変換は図 15 のようになる．ここで，vmK に対しては vmKstate に対するコンパイラと逆コンパイラを融合している．この変換により，H.vm では Var n であった部分 (図 6 の下線部) が L.vm ではコンパイル後の形である [Access n](図 13 の下線部) に変わっている．

相互再帰している H.vmK と補助的に使われている H.shiftE も同様の手続きで低水準化できる．

また，最終的に L.vm には逆コンパイラである decomp やそれを使った decomp_val などの関数は出現していない．これは融合変換の過程で compiler と打ち消し合ったからである．このため，decomp の定義を明示的に書く必要はない．

5 おわりに

5.1 関連研究

Ager らはインタプリタのプログラムを変換していくことでコンパイラと仮想機械を実装する手法を提案した [1].

Igarashi と Iwaki は，Ager らの手法を応用し，$\lambda^{\bigcirc}$ のインタプリタから，プログラム変換により，コンパイラと仮想機械プログラムを導出した [4]．これらのプログラムは 3 章で紹介した．本論文で導出した仮想機械は，彼らが導出した仮想機械が元になっている．

木谷と浅井は，Ager らの手法を応用し，shift/reset を含む λ 計算インタプリタから，コンパイラと仮想機械を導出した [6]．この手法によって導出されるコンパイラは一つの項の構成子を複数の命令の列にコンパイルしていて，より基本的な命令セットになっているといえる．本研究の手法はこの手法で導出された仮想機械にも適用できる．

Danvy と Millikin [2] は，Ohori と Sasano の関数融合変換を用いて，small-step semantics に基づく抽象機械 (インタプリタ) と big-step semantics に基づく抽象機械の等価性を示している．融合変換により言語処理系間の等価性を示すもうひとつの例であるが，本研究とは融合変換を使う目的が異なっている．

5.2 まとめと今後の課題

本論文では低水準コード生成を行う仮想機械の系統的な導出方法を示した．この方法で導出された仮想機械は [4] で示されたものと一致している．

今後の課題としては，以下のようなものが考えら

```
let decomp_vmstate = function
    ([Access n], 0, e, k) -> ([Access n], 0, decomp_env e, decomp_cont k)
  | ([Access n], lv, e, k) -> ([Access n], lv, decomp_env e, decomp_cont k)
  ⋮
  | (Enter::c0, 0, e, k) -> (Enter :: c0, 0, decomp_env e, decomp_cont k)
  | (Enter::c0, lv, e, k) -> (Enter :: c0, lv, decomp_env e, decomp_cont k)
  ⋮
```

図 11　逆コンパイラを使ったデータ構造変換関数

```
(* comp_Hvm : H.vmstate -> L.value *)
let rec comp_Hvm = function
    ([Access n], 0, e, k) -> comp_HvmK (k, List.nth e n)
  | ([Access n], lv, e, k) -> comp_HvmK (k, H.Q (Var n))
  | ([Close c0], 0, e, k) -> comp_HvmK (k, H.Clos (e, c0))
  | ([Close c0], lv, e, k) -> comp_Hvm (c0, lv, H.shiftE(e,lv), H.QAbs k)
  | (Push c1::c0, 0, e, k) -> comp_Hvm (c0, 0, e, H.EvArg (c1, e, k))
  | (Push c1::c0, lv, e, k) -> comp_Hvm (c0, lv, e, H.QApp' (c1, lv, e, k))
  | (Enter::c0, 0, e, k) -> comp_Hvm (c0, 1, e, H.Quote k)
  | (Enter::c0, lv, e, k) -> comp_Hvm (c0, lv + 1, e, H.QNext k)
  | (Leave::c0, 1, e, k) -> comp_Hvm (c0, 0, e, H.Unquote k)
  | (Leave::c0, lv, e, k) -> comp_Hvm (c0, lv - 1, e, H.QPrev k)
```

図 12　comp_val ∘ H.vm

```
module L
  let rec vm = function
      ([Access n], 0, e, k) -> vmk (k, List.nth e n)
    | ([Access n], lv, e, k) -> vmk (k, Q [Access n])
    | ([Close c0], 0, e, k) -> vmk (k, Clos (e, c0))
    | ([Close c0], lv, e, k) -> vm (c0, lv, shiftE (e, lv), QAbs k)
    | (Push c1::c0, 0, e, k) -> vm (c0, 0, e, EvArg (c1, e, k))
    | (Push c1::c0, lv, e, k) -> vm (c0, lv, e, QApp' (c1, lv, e, k))
    | (Enter::c0, 0, e, k) -> vm (c0, 1, e, Quote k)
    | (Enter::c0, lv, e, k) -> vm (c0, lv + 1, e, QNext k)
    | (Leave::c0, 1, e, k) -> vm (c0, 0, e, Unquote k)
    | (Leave::c0, lv, e, k) -> vm (c0, lv - 1, e, QPrev k)
```

図 13　L.vm：低水準コード生成を行う仮想機械

```
comp_Hvm (Enter :: c0, 0, decomp_env e, decomp_cont k)
↓ (comp_Hvm のインライン展開)
comp_Hvm (c0, 1, decomp_env e, H.Cont3 (decomp_cont k))
↓ (恒等関数 decomp_vmstate∘comp_vmstate の挿入)
comp_Hvm (decomp_vmstate (comp_vmstate (c0, 1, decomp_env e, H.Cont3 (decomp_cont k))))
↓ (comp_vmstate のインライン展開)
comp_Hvm (decomp_vmstate (c0, 1, comp_env (decomp_env e), comp_cont (H.Cont3 (decomp_cont k))))
↓ (comp_Hvm ∘ decomp_vmstate の再帰呼び出しによる置き換えと comp_cont のインライン展開)
vm (c0, 1, comp_env (decomp_env e),  L.Cont3 (comp_cont (decomp_cont k)))
↓ (comp と decomp の打ち消し)
vm (c0, 1, e, L.Cont3 k)
```

図 14　融合変換の過程 (1)

```
comp_Hvm ([Access n], lv, decomp_env e, decomp_cont k)
↓ (comp_Hvm のインライン展開)
comp_HvmK (decomp_cont k, H.Q (Var n))
↓ (恒等関数 decomp_vmKstate∘comp_vmKstate の挿入)
comp_HvmK (decomp_vmKstate (comp_vmKstate (decomp_cont k, H.Q (Var n))))
↓ (comp_vmKstate のインライン展開)
comp_HvmK (decomp_vmKstate (comp_cont (decomp_cont k), comp_val (H.Q (Var n))))
↓ (comp_HvmK ∘ decomp_vmKstate の再帰呼び出しによる置き換え)
vmK (comp_cont (decomp_cont k), comp_val (H.Q (Var n)))
↓ (comp と decomp の打ち消し)
vmK (k, comp_val (H.Q (Var n)))
↓ (comp_val のインライン展開)
vmK (k, L.Q [Access n])
```

図 15　融合変換の過程 (2)

れる.

- この手法を適用するために必要な仮想機械やコンパイラの性質について調べる
- より基本的な命令セットで動く仮想機械に対しこの手法を適用する

参 考 文 献

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: A functional derivation. Technical Report RS-03-14, BRICS, March 2003.

[2] Olivier Danvy and Kevin Millikin. A simple application of lightweight fusion to proving the equivalence of abstract machines. Technical Report RS-07-8, BRICS, March 2007.

[3] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings of IEEE Symposium on Logic In Computer Science (LICS'96)*, pp. 184–195, July 1996.

[4] Atsushi Igarashi and Masashi Iwaki. Deriving compilers and virtual machines for a multi-level language. In Zhong Shao, editor, *Proceedings of 5th Asian Symposium on Programming Languages and Systems (APLAS 2007)*, Vol. 4807 of *Lecture Notes in Computer Science*, pp. 206–221, Singa-

pore, November/December 2007. Springer-Verlag.

[5] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL2007)*, pp. 143–154, January 2007.

[6] 木谷有沙, 浅井健一. プログラム変換によるインタプリタからコンパイラの導出. 第 12 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 (PPL2010) 論文集, pp. 206–220, March 2010.