仮想機械導出のためのプログラム変換

木谷有沙　(指導教官: 浅井健一)

1　はじめに

継続とは，プログラム実行中のある時点における，残りの計算のことを指す．継続を処理することは，プログラムの制御フローを操作することに等しい．継続処理オペレータがあれば，例外処理のような挙動をユーザが定義することや，型の考慮された大域ジャンプをすることが可能になる．しかし現在，継続処理を実装しているプログラミング言語は多くない．ユーザが機械語実装するにしても，継続処理にともなうスタックやヒープのコピー操作は自明ではなく，困難である．

そこで本研究では，shift/reset を定義する λ 計算インタプリタから，それと全く同じ挙動を示すことが保証されているコンパイラ及び仮想機械を導出する．

そのために，変換前後の評価器が等価なものであることが保証されたプログラム変換をインタプリタに施していき，低レベルな機械に近づけて行く．これは Danvy ら [3] の手法と同じ考え方だが，我々は彼らよりも実際の機械語実装に近い仮想機械の導出を目指し，スタック導入，環境退避，戻り番地退避などの変換を新しく導入している．そうして得られた評価器を，Ager [1] らの手法に基づいてコンパイラと仮想機械へと分割する．

2　shift / reset

shift/reset とは，限定継続を処理するオペレータであり，CPS インタプリタにおいて，その定義が与えられている [2]．本研究では，`shift(...)` と書くと括弧内の関数に現在の継続が渡され，`reset(...)` と書くと括弧内の `shift` で取って来られる継続が括弧内の処理のみに限定される．

3　プログラム変換

本研究では，call-by-value の形無し λ 計算を以下のように shift/reset で拡張したものを対象とする．

$$t \quad ::= \quad x \mid \lambda x.\, t \mid t_0\ t_1 \mid \texttt{shift}(t) \mid \texttt{reset}(t)$$

変数，関数定義，関数呼び出しの他に shift と reset を記述できる．

値としては，関数定義の際に生成されるクロージャと，shift オペレータによって取り出される継続の二つがある．

$$v \quad ::= \quad [\lambda(v, c).t] \mid [c]$$

これらを関数型言語 OCaml を用いて実装した評価器に，以下のプログラム変換をかける．

3.1　スタック導入

定義の通り実装された評価器では，継続の中で使われる計算の中間値は自由変数になる．本研究では変換の結果，継続はコードポインタの受け渡しにしたいので，計算の中間値が継続に含まれていては困るし，Ager らの手法を用いるためには継続から自由変数が取り除かれている必要がある．よってここでは，新しくスタックに相当する引数を評価器に導入し，計算の中間値はスタックに積んで受け渡すよう変換する．スタック導入は我々が提案し，変換前後の評価器の等価性を証明した変換である．

3.2　環境退避

継続の中で使われる環境（変数値リスト）も，まだ自由変数になっている．計算の中間値と同様，これも継続に含まれるのは避けたい．よって，環境は評価器の引数としてではなく，スタックの先頭に積んで受け渡すように変換する．環境退避は我々が提案し，変換前後の評価器の等価性を証明した変換である．

3.3　戻り番地退避

継続の中に含まれる継続も自由変数になっている．この継続は低レベルな実装における戻り番地に相当する．これもスタックに積むように変更すれば，継続から完全に自由変数を取り除くことができる．戻り番地退避は我々が提案した変換であり，変換前後の評価器の等価性は，環境退避と同様の方法ですぐに検証できる．

4　コンパイラ及び仮想機械の導出

Ager らの手法に則り，以下の手順で評価器を分割し，コンパイラと仮想機械を得る．

4.1　カリー化

評価器をカリー化し，項だけ受け取れば処理できる部分とそれ以降に分割する．前者がコンパイラ，後者が仮想機械の処理に相当する．カリー化前後の評価器の等価性は保証されている．

4.2　関数合成に分解

カリー化の結果，評価器の出力は `fun` 文になっているが，それぞれの `fun` 文で行う処理に名前を付けて関数（自由変数を含まない関数，コンビネータ）として定義し，評価器はこれらの関数を呼び出すことにする．前節の変換で評価器から正しく自由変数が取り除かれていれば，各処理はここで名付けた関数と再帰呼び出しの合成として書き換えられる．この各関数が仮想機械の命令に相当すると考えると，評価器が仮想機

$$
\begin{array}{rcl}
il & \Rightarrow & \langle il, \mathtt{VEnv}([]) :: \mathtt{VK}([]) :: [] \rangle \\
\hline
\langle \mathtt{IPushEnv} :: il', \mathtt{VEnv}(vs) :: s \rangle & \Rightarrow & \langle il', \mathtt{VEnv}(vs) :: \mathtt{VEnv}(vs) :: s \rangle \\
\langle \mathtt{IPopEnv} :: il', v :: \mathtt{VEnv}(vs) :: s \rangle & \Rightarrow & \langle il', \mathtt{VEnv}(vs) :: v :: s \rangle \\
\langle \mathtt{IPushK}(il) :: il', v :: \mathtt{VEnv}(vs) :: s \rangle & \Rightarrow & \langle il', \mathtt{VEnv}(vs) :: \mathtt{VK}(il) :: s \rangle \\
\langle \mathtt{IAccess}(n) :: \_, \mathtt{VEnv}(vs) :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il', (List.nth\ vs\ n) :: s \rangle \\
\langle \mathtt{IPushCls}(il) :: \_, \mathtt{VEnv}(vs) :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il', \mathtt{VFun}(il, vs) :: s \rangle \\
\langle \mathtt{ICall} :: \_, \mathtt{VFun}(il, vs) :: v :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il, \mathtt{VEnv}(v :: vs) :: \mathtt{VK}(il') :: s \rangle \\
\langle \mathtt{ICall} :: \_, \mathtt{VCont}(s', il'') :: v :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il, \langle il'', v :: s' @ s \rangle \rangle \\
\langle \mathtt{IShift} :: \_, \mathtt{VFun}(il, vs) :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il, \mathtt{VEnv}(\mathtt{VCont}(s, il') :: vs) :: \mathtt{VK}(id) :: [] \rangle \\
\langle \mathtt{IShift} :: \_, \mathtt{VCont}(s', il'') :: v :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il'', \mathtt{VCont}(s, il') :: s' \rangle \\
\langle \mathtt{Reset}(il) :: \_, \mathtt{VEnv}(vs) :: \mathtt{VK}(il') :: s \rangle & \Rightarrow & \langle il', \langle il, \mathtt{VEnv}(vs) :: \mathtt{VK}([]) :: [] \rangle @ s \rangle \\
\hline
\langle [], v :: \_ \rangle & \Rightarrow & \langle v \rangle
\end{array}
$$

図 1: 仮想機械から導かれる状態遷移規則

械の命令列を出力していると見なせるようになる．関数を展開すれば，もとの評価器と全く同じものになることより，この変換前後の評価器の等価性は保証されている．

4.3 ファンクタとモジュールで実装

項を受け取ったら関数を返す部分をファンクタ，各関数の処理は別のモジュールとして実装する．モジュールを展開すれば前節と全く同じ評価器が得られることより，この変換前後の等価性は保証されている．

4.4 コンパイラと仮想機械

同じファンクタを使って，今度はランタイム処理（スタックを受け取って行う処理）をせずに命令列を出力する，つまりコンパイルだけするモジュールを実装する．そして，そのコンパイル結果を受け取りランタイム処理を行う仮想機械をさらに別のモジュールとして実装する．こうして得られたコンパイラ及び仮想機械の定義は以下のようになる．

- 入力項と仮想機械の命令列

$$
\begin{array}{rcl}
t & ::= & x \mid \lambda x.\, t \mid t_0\ t_1 \mid \mathtt{shift}(t) \mid \mathtt{reset}(t) \\
i & ::= & \mathtt{IPushEnv} \mid \mathtt{IPopEnv} \mid \mathtt{IPushK}(il) \\
 & & \mid \mathtt{IAccess}(n) \mid \mathtt{IPushCls}(il) \mid \mathtt{ICall} \\
 & & \mid \mathtt{IShift} \mid \mathtt{IReset}(il) \\
il & ::= & i :: il
\end{array}
$$

- コンパイラ

$$
\begin{array}{rcl}
[\![x, xs]\!] & = & [\mathtt{IAccess}(get(x, xs))] \\
[\![\lambda x.\ t,\ xs]\!] & = & [\mathtt{IPushCls}([\![t, x :: xs]\!])] \\
[\![t_0\ t_1,\ xs]\!] & = & \mathtt{IPushEnv} \\
 & & :: \mathtt{IPushK}(\mathtt{IPopEnv} \\
 & & :: \mathtt{IPushK}([\mathtt{ICall}]) \\
 & & :: [\![t_0,\ xs]\!]) :: [\![t_1,\ xs]\!] \\
[\![\mathtt{shift}(t),\ xs]\!] & = & \mathtt{IPushK}([\mathtt{IShift}]) :: [\![t,\ xs]\!] \\
[\![\mathtt{reset}(t),\ xs]\!] & = & [\mathtt{IReset}([\![t,\ xs]\!])]
\end{array}
$$

- 値

$$
\begin{array}{rcl}
v & ::= & [il, vs] \mid [s, il] \mid [vs] \mid [il] \\
s & ::= & v :: s \\
vs & ::= & v :: vs
\end{array}
$$

- 状態遷移規則

  仮想機械の状態遷移規則は図 1 のように定義される．$\langle\ \rangle$ の中に $\langle\ \rangle$ が含まれる場合は，内側の $\langle\ \rangle$ から先に評価を行う．各状態に対して適用される状態遷移規則は決定的に定められる．

この実装自体は 4.3 節で作ったモジュールの非関数化であり，非関数化の正当性より，この変換前後の評価器の等価性は保証されているといえる．

導出の結果得られた仮想機械では，継続処理のコードポインタの受け渡し及びスタックコピーとしての実装をモデル化できている．また，処理が値まで落ちたら，スタックに積まれた戻り番地を読み取りジャンプするという挙動になっている．これは低レベルな機械での call/return の振る舞いを模倣しているといえる．

5 まとめと今後の課題

本研究では，shift/reset を含む λ 計算のインタプリタに対してスタック導入，環境退避，戻り番地退避という変換を施し，より低レベルな機械語実装に近く，継続から自由変数が取り除かれた評価器を得た．その評価器に Ager らの手法を用いて，コンパイラと仮想機械を得た．変換前後の評価器の等価性が保証されている変換のみを用いているため，本研究で得たコンパイラと仮想機械は，もともとのインタプリタと全く同じ挙動をすることが保証されている．かつ，このコンパイラと仮想機械は実際の機械語実装における call/return に近い振る舞いを模倣できている上，shift/reset の処理をコードポインタの受け渡し及びスタックのコピーとしてモデル化できている．今後はこの仮想機械と実際の機械語の命令セットとの対応をより明らかにし，shift/reset の機械語実装の正当性を保証するものにしたい．

参考文献

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. From interpreter to compiler and virtual machine: A functional derivation. Technical report, BRICS, RS-03-14, 2003.

[2] O. Danvy and A. Fillinski. A functional abstraction of typed contexts. Technical report, DIKU, 89/12, 1989.

[3] O. Danvy and K. Millikin. A rational deconstruction of landin's secd machine with the j operator. *Logical Methods in Computer Science*, Vol. 4, 4:12, pp. 1–67, 2008.