限定継続演算子の仮想機械導出

藤井 舞花　(指導教員：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である。例えば `1 + 2 * 3` という計算があり `2 * 3` の時点での残りの計算は `1 + [.]` である。この `1 + [.]` が継続である。

限定継続演算子は計算の範囲を指定しその継続の切り取りや、複製をすることができる。継続の取り出しやコピーはプログラムの制御フローを操作することに相当し、継続を処理するオペレータがあれば、大域ジャンプができるようになる。つまりユーザー自身が自由に定義できる例外処理のような挙動が可能となる。

しかし、継続処理を直接実装するプログラミング言語は少ない。ほとんどの場合 CPS 変換による意味論が与えられているだけで、データスタックやヒープへのコピーの挙動まで説明しているものは数える程である。

本研究では、低レベルな実装を目指して、限定継続演算子に対するインタプリタに、各種正当性の保証されたプログラム変換を施し、最終的にコンパイラと仮想機械の導出を行う。それを通して、低レベルな言語に近づけた実装の指針を示す。これまで shift / reset のみに対しては仮想機械が得られていた [1] が、今回 4 種類の限定継続演算子全てに拡張した。

紙面の関係からインタプリタのコードを載せることはできなかった。詳しくは付随するコードを参照せよ。

2　限定継続演算子

限定継続演算子には shift / reset [3], control / prompt [5]、shift0 / reset0 [8]、control0 / prompt0 [6] がある。shift, control, shift0, control0 は現在の継続を切り取る演算子で、reset, prompt, reset0, prompt0 は切り取る継続の範囲を限定する演算子である。継続を切り取る演算子らには、切り取られた継続が呼び出された時の「現在の継続」が呼び出された中から見えるかどうか、また、継続が切り取られた時に継続の範囲を限定した演算子の「一つ外側の継続」が見えるかどうかに違いがある。これらの違いを表現するため、前者に対して trail、後者に対してメタ継続というものがインタプリタでは導入されている。継続の範囲を限定する演算子はそれらの挙動に違いがないため、本研究では reset のみを扱う。限定継続演算子の詳しい理解は [4] を参照されたい。

3　インタプリタの型定義

本研究では、型なし λ 計算を限定継続演算子で拡張した構文を持つ言語を対象とする。これを関数型言語 OCaml を用いて実装した。インタプリタの項と値の定義は図 1 のようになる。ベースとするインタプリタは Shan [10] によるものである。これは継続を保持する trail をリストなどのデータ構造に具体化せず、関数として外延的に表現したインタプリタである。メタ継続は継続と trail のリストとなっている。このインタプリタに対してプログラム変換をして仮想機械を導出していく。

```
(* Syntax *)
type e = Var of string | Fun of string * e
       | App of e * e | Reset of e
       | Shift of string * e
       | Control of string * e
       | Shift0 of string * e
       | Control0 of string * e
(* Value *)
type v = VFun of (v -> c -> t -> m -> v)
and  c = v -> t -> m -> v
and  t = TNil | Trail of c
and  m = MNil | MCons of (c * t) * m
```

図 1: 項と値の型定義

4　プログラム変換

限定継続演算子を含む仮想機械の導出の手順を示す。4.1 節から 4.4 節 では、インタプリタからスタックのデータ構造を抽出する。また、4.5 節から 4.9 節 で、それをコンパイラと仮想機械に分離する。

4.1　非関数化

スタックを導入するため、まず継続など高階関数を 1 階のデータに変換する非関数化 [9] を行う。非関数化とはプログラム中に出現する高階関数（OCaml の fun 文、λ 式）に対して関数の body 部分で自由変数となっているものを保持するコンストラクタを定義し、高階関数の代わりにそのコンストラクタを使用することである。特に継続は、初期継続、次への継続が格納されたもの、継続同士を合わせたようなものからなるデータ構造になる。

4.2　継続を線形リスト化

非関数化された継続は空リスト、cons、append であるようなリスト構造とみなすことができるため、線形化する。前節で保持されていた継続以外のデータを格納するためフレームという構造を用意し、継続はそのフレームのリストとなった。ここで、継続の型に対して OCaml の実際のリストを用いると、合わせた継続同士の区切りが分からなくなってしまうため、リストを表すようなコンストラクタを用いている。

4.3　スタック導入

前節で導入したフレームには、変換時のデータ（static データ）と実行時のデータ（dynamic データ）の両方が含まれている。本研究の目的はインタプリタをコンパイラと仮想機械に変換することなので、この二種類のデータを分ける変換を行う。そこで実行時のデータを格納するスタックを導入する。フレームには変換時のデータのみを残し、それに対応する実行時のデータはスタックの中に置くようにした。

4.4　非線形化、関数化

4.1 節の非関数化と 4.2 節の継続の線形化は、スタックを導入するために施した変換である。4.3 節でスタックが導入できたので、継続を非線形化、関数化し元の状態に戻す。初めのインタプリタと比較すると、値の保存と復元をモデル化したインタプリタとなった。

$$
\begin{array}{rcl}
c & \Rightarrow & \langle c,\ [VEnv([])],\ [],\ [] \rangle \\
\hline
\langle [],\ v :: [],\ [],\ [] \rangle & \Rightarrow & \langle v \rangle \\
\langle [],\ v :: [],\ [],\ ((c,\ s) :: t) :: m \rangle & \Rightarrow & \langle c,\ v :: s,\ t,\ m \rangle \\
\langle [],\ v :: [],\ (c,\ s) :: t,\ m \rangle & \Rightarrow & \langle c,\ v :: s,\ t,\ m \rangle \\
\langle IAccess\ (n) :: c,\ VEnv\ (vs) :: s,\ t,\ m \rangle & \Rightarrow & \langle c,\ (List.nth\ vs\ n) :: s,\ t,\ m \rangle \\
\langle IPush\_closure\ (c') :: c,\ VEnv\ (vs) :: s,\ t,\ m \rangle & \Rightarrow & \langle c,\ VFun\ (c',\ vs) :: s,\ t,\ m \rangle \\
\langle IReturn :: \_,\ v :: VK(c) :: s,\ t,\ m \rangle & \Rightarrow & \langle c,\ v :: s,\ t,\ m \rangle \\
\langle IPush\_env :: c,\ VEnv\ (vs) :: s,\ t,\ m \rangle & \Rightarrow & \langle c,\ VEnv\ (vs) :: VEnv\ (vs) :: s,\ t,\ m \rangle \\
\langle IPop\_env :: c,\ v :: VEnv\ (vs) :: s,\ t,\ m \rangle & \Rightarrow & \langle c,\ VEnv\ (vs) :: v :: s,\ t,\ m \rangle \\
\langle ICall :: c,\ v :: VFun\ (c',\ vs) :: s,\ t,\ m \rangle & \Rightarrow & \langle c',\ VEnv\ (v :: vs) :: VK\ (c) :: s,\ t,\ m \rangle \\
\langle ICall :: c,\ v :: VContS\ ((c',\ s') :: t') :: s,\ t,\ m \rangle & \Rightarrow & \langle c',\ v :: s',\ t',\ ((c,\ s) :: t) :: m \rangle \\
\langle ICall :: c,\ v :: VContC\ ((c',\ s') :: t') :: s,\ t,\ m \rangle & \Rightarrow & \langle c',\ v :: s',\ t' \ @\ (c,\ s) :: t,\ m \rangle \\
\langle IShift\ (c') :: c,\ VEnv\ (vs) :: s,\ t,\ m \rangle & \Rightarrow & \langle c',\ VEnv\ (VContS\ ((c,\ s) :: t) :: vs) :: [],\ [],\ m \rangle \\
\langle IControl\ (c') :: c,\ VEnv\ (vs) :: s,\ t,\ m \rangle & \Rightarrow & \langle c',\ VEnv\ (VContC\ ((c,\ s) :: t) :: vs) :: [],\ [],\ m \rangle \\
\langle IShift0\ (c') :: c,\ VEnv\ (vs) :: s,\ t,\ ((c_0,\ s_0) :: t_0) :: m_0 \rangle & \Rightarrow & \langle c' \ @\ c_0,\ VEnv\ (VContS\ ((c,\ s) :: t) :: vs) :: s_0,\ t_0,\ m_0 \rangle \\
\langle IControl0\ (c') :: c,\ VEnv\ (vs) :: s,\ t,\ ((c_0,\ s_0) :: t_0) :: m_0 \rangle & \Rightarrow & \langle c' \ @\ c_0,\ VEnv\ (VContC\ ((c,\ s) :: t) :: vs) :: s_0,\ t_0,\ m_0 \rangle \\
\langle IReset\ (c') :: c,\ VEnv\ (vs) :: s,\ t,\ m \rangle & \Rightarrow & \langle c',\ VEnv(vs) :: [],\ [],\ ((c,\ s) :: t) :: m \rangle
\end{array}
$$

図 2: 仮想機械の状態遷移規則

4.5 環境退避

これまで引数として渡されていた、値や値の環境、スタックは常に共に現れる。仮想機械では、値の受け渡しは全てスタック経由で行いたいので、値は常にスタック上で受け渡すように変換する。評価器に渡されていた値の環境はスタックにまとめて渡している。

4.6 命令に分解

インタプリタを、「static データのみを受け取って処理する部分」と「dynamic データを受け取ったら処理する部分」に分割する。前者がコンパイラ、後者が仮想機械となる。この分解は、これまでのインタプリタの型を static データと、dynamic データであるその後の引数に分けることで行う。まず、dynamic データを受け取ったら行う仕事を表す「命令」を導入する。そして、評価器は項と変数名の環境の static データのみを処理するようにし、評価器の出力を命令とする。その後の仕事を表す命令は関数として、dynamic なデータを用いるこれまでの挙動を定義している。この変換はこれまでの評価器で行われていたことを単に命令群として関数定義しているだけで、中身を展開したら 4.5 節のインタプリタに戻る。

4.7 非関数化

前節までで導入した命令をより機械語に近づけたものにするため、高階関数になっているものを 4.1 節 と同様にして非関数化する。非関数化すると、命令は各関数内で作られていた高階関数に従って、対応する構成子が導出される。これが仮想機械の命令となる。さらに、継続は 4.2 節 と同様の構造をした命令のリストとして表されるようになった。

4.8 命令を線形化

評価器が出力する命令列は木構造をなしているが、この木構造は高々一つの子要素しか持たないため線形リスト化できる。変換後 shift0 と control0 では「reset の一つ外側の継続」を追跡するため、継続（命令列）の append が必要な箇所があることが分かった。

4.9 trail を線形化

前節まででは trail に継続を追加する際関数合成を行なっていたが、後に実行するべき継続をその都度、順に積むようにした。つまり trail を継続とスタックのペアのリストにした。こうすることで継続は append を表す構成子が不要となり、OCaml の実際のリストで表されるようになった。

5 限定継続演算子の仮想機械

以上の変換を施したプログラムから仮想機械を導出した。仮想機械の状態遷移規則は図 2 のように定義される。標準呼び出し規則だけでなく、継続の切り取りや呼び出しでのデータスタックのコピーの動作をモデル化している。継続全てを `VK` 構成子によって戻り番地の値と考えると、trail やメタ継続は結局スタックのリストと捉えることができる。状態において s, t, m 部分に注目すると、どの命令が呼び出されてもこれら三つの順番は変化していないことが分かる。各々の間に区切り記号を用意すればコピー操作が不要となり、単に区切り記号を変更すれば良いということを示している。よって全て繋げて一本のスタックとみなせる。

6 まとめ

本研究では λ 計算に限定継続演算子を拡張したインタプリタからコンパイラと仮想機械を導出した。得られた仮想機械は限定継続演算子の実装手法を提供する。具体的にはデータスタックの挙動や、ヒープへのコピーをするべき箇所を示している。今後の課題として、限定継続演算子の他にも継続を明示的に扱えるものとして algebraic effect handlers [2, 7] が盛んに研究されおり、この実装手法についても導出していきたい。

参考文献

[1] Kenichi Asai and Arisa Kitani. Functional derivation of a virtual machine for delimited continuations. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pp. 87–98, 2010.

[2] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, Vol. 84, No. 1, pp. 108–123, 2015.

[3] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pp. 151–160, 1990.

[4] R Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, Vol. 17, No. 6, pp. 687–730, 2007.

[5] Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pp. 180–190, New York, NY, USA, 1988. ACM.

[6] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A Generalization of Exceptions and Control in ML-like Languages. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pp. 12–23, New York, NY, USA, 1995. ACM.

[7] Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect Handlers via Generalised Continuations. *Journal of Functional Programming*, Vol. 30, , 2020.

[8] Marek Materzok and Dariusz Biernacki. Subtyping Delimited Continuations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pp. 81–93, New York, NY, USA, 2011. ACM.

[9] John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference-Volume 2*, pp. 717–740, 1972.

[10] Chung-chieh Shan. A static simulation of dynamic delimited control. *Higher-Order and Symbolic Computation*, Vol. 20, No. 4, pp. 371–401, 2007.