代数的効果を含むプログラムのステップ実行

理学専攻・情報科学コース 1840669 古川 つきの　(指導教員：浅井 健一)

1　はじめに

ステッパとはプログラミング教育やデバッグのために使うツールであり，プログラムが代数的に書き換わる様子を出力することで実行過程を見せるものである．例えば OCaml のプログラム `let a = 1 + 2 in 4 + a` を入力すると図 1 のようなステップ表示をする．

```
Step 0:  let a = (1 + 2) in (4 + a)
Step 1:  let a = 3 in (4 + a)
Step 1:  let a = 3 in (4 + a)
Step 2:  4 + 3
Step 2:  4 + 3
Step 3:  7
```

図 1: `let a = 1 + 2 in 4 + a` の実行ステップ

これまでに Racket [1] の教育用に制限した構文や OCaml [3, 7] などを対象にステッパが作られてきたが，shift/reset [2] や 代数的効果 [5] (以下，algebraic effects) といった，継続を明示的に扱う機能を含む言語のステッパは作られていない．継続を扱うプログラムの挙動を理解するのは困難なので，そういった言語に対応したステッパを作ることが本研究の目的である．

ステッパは簡約のたびにその時点でのプログラム全体を出力するインタプリタなので，実行している部分式のコンテキスト (周りの式) の情報が常に必要になる．継続を扱うような複雑な機能を持つ言語を対象にしたステッパでは，コンテキストがどのような構造をしているかが自明でない．そこで，通常のインタプリタ関数をプログラム変換することで機械的にコンテキストの情報を保持させ，ステッパを実装する方法を示す．実際に型無し λ 計算と algebraic effects から成る言語を対象にしてステッパを実装する過程を説明する．

2　algebraic effects

algebraic effects は，例外や状態などの副作用を表現するためのプログラミング言語機能である．型無し λ 計算に algebraic effects を足した言語を図 2 のように定義できる．

「オペレーション」に引数を渡すとそのオペレーションの「エフェクト」が実行されるというのが algebraic effects の基本的な動作である．オペレーションのエフェクトは「ハンドラ」で定義することができ，使用するハンドラを変えることでオペレーションに別のエフェクトを持たせることができるのが特徴である．

エフェクトの定義においては，オペレーションに渡された引数 `x` の他に「継続」を束縛した変数 `k` を使用することができる．継続とは，プログラム実行のある時点での残りの計算のことであり，例えば `1 + (2 + 4)` という式の `2 + 4` を計算している時点での継続は「今の計算の結果を 1 に足す」という計算である．特に，残りの計算の全てではなく一部のみの継続を限定継続という．

```
v :=                    (値)
     x                  変数
   | fun x -> e         関数
   | fun x => e         継続
e :=                    (式)
     v                  値
   | e e                関数/継続適用
   | op e               オペレーション呼出
   | with h handle e    ハンドル
h :=                    (ハンドラ)
    {return x -> e;     正常終了処理の定義
     op(x; k) -> e;     エフェクト定義 (0 個以上)
     ...;
     op(x; k) -> e}
```

図 2: 構文の定義

algebraic effects のエフェクトで使用できる継続 `k` はオペレーション呼び出しをしたところからそのオペレーションを定義している with handle 文までの限定継続である．オペレーションが呼び出されたら，そのオペレーションを定義している with handle 文までの限定継続を変数 `k` に束縛する．そのため，algebraic effects では with handle 文までの限定継続をひとまとまりとして扱う．

本研究では，まず algebraic effects を定義するインタプリタを与えている．継続適用をインタプリタのメタ継続の適用によって実現するためにインタプリタが継続を持つ必要があるので，with handle 文の内部の実行を基本的に CPS (継続渡し形式) で実装した．

3　ステッパの実装とコンテキスト

ステッパはインタプリタの一種であるため，通常のインタプリタ関数に機能を付け加えることで実装することができる．

ところが，通常のインタプリタの構造では，書き換わる部分以外の情報を参照できないという問題がある．図 1 の例では，インタプリタは再帰的に部分式を実行し，最初に計算を進められる部分である `(1 + 2)` を計算して，値 `3` を得る．その時にステッパは図 1 の最初の 2 行を出力したいが，`(1 + 2)` を再帰的に実行している最中なので，周りの式 `let a = [.] in (4 + a)` の情報はスタックにしかなく，出力することができない．この周りの式のことをコンテキストと呼ぶ．

つまり，ステッパを実装するためにはコンテキストの情報が必要なので，その情報を得るためにインタプリタプログラムが全体の中のどこを実行しているのかという情報を明示的に扱えるようにしたい．我々は以前 [3]，コンテキストを表すデータ型を定義し，インタプリタ関数の再帰呼び出し時の構造に合わせて引数として渡すようにすることで，コンテキストの情報を明示的に保持させるようにした．明示的に扱えるコンテ

キストの情報があればその段階でのプログラム全体を再構成できるので，それを出力することでステッパを実装することができた．

しかしコンテキストを表すデータ型は言語ごとに定義する必要があり，その構造は自明なものとは限らない．そこで本研究では，通常のインタプリタに機械的な変換を施すことでコンテキストの情報を持つインタプリタおよびコンテキストのためのデータ型を導出し，それを利用してステッパを実装する方法を提案する．

4 インタプリタの変換

本節では，CPS (継続渡し形式) を使って書かれた通常のインタプリタに，正当性が証明されている (プログラムの動作を変えない) 非関数化 [6] と CPS 変換 [4] という 2 種類のプログラム変換を施すことで，コンテキストを明示的に保持するインタプリタを得てそこからステッパを導出する方法について順に説明する．

4.1 非関数化

非関数化は，高階関数を 1 階のデータ構造で表現する方法である．この変換は機械的に行うことができる．

前節で書いたように，我々が定義したインタプリタは with handle 文の内部の実行を CPS を用いて実装している．つまりそれぞれの with handle 文の内部を実行している間，その with handle 文までの限定継続を関数として引数に保持している．その関数を非関数化することで，with handle 文までの継続を，内容の参照が可能なデータとして扱うことができるようになる．

4.2 CPS 変換

CPS 変換はプログラムの継続を引数として明示的に持つようにするプログラム変換であり，これも機械的に行うことができる．

非関数化したインタプリタ関数を CPS 変換すると，インタプリタの実行全体のメタ継続関数を引数に持つようになる．このメタ継続は入力プログラムでの with handle 文から外の継続に対応している．

4.3 非関数化

CPS 変換したことで引数に全体の継続を関数として持つようになったので，これも非関数化すると，with handle から外の継続を表すデータを得ることができる．1 度目の非関数化で with handle までの継続のデータも得ているので，ここまでの変換で継続を明示的に保持しているインタプリタが導出できた．

4.4 出力

ここまでの変換によってコンテキストの情報が得られたので，プログラム全体を再構成して出力することが可能になった．プログラムの計算が進んで書き換わる部分で書き換え前後のプログラムを出力するように変更するとステッパが実装できる．

4.5 CPS インタプリタを基にしたステッパ

4.4 節でステッパが実装できたが，機械的な変換を何度も施したことでステッパ関数は複雑になっており，保守性に欠けるなど実用上の問題がある．そこで，元の CPS インタプリタ関数内の対応する部分に出力に必要なコンテキスト情報および出力命令を追加すると，CPS インタプリタと同じ構造でコンテキスト情報のための引数を増やしたようなステッパ関数を得られる．

以上の変換によって，CPS インタプリタからコンテキストを表せるデータ構造および再帰呼び出し時に渡すべきコンテキスト情報を導出し，ステッパを実装することができた．

5 他の言語への対応

4 節と同様の変換によって，他のいくつかの言語に対するステッパの導出を試みた．型無し λ 計算を対象言語とすると，コンテキストを区切るような制御が無いため，CPS インタプリタを非関数化するのみで全てのコンテキスト情報を保持することができ，ステッパが実装できた．例外処理などのための構文 try-with および shift/reset を含む言語を対象にすると，algebraic effects の場合と全く同じ手順でステッパを実装することができた．元のインタプリタが CPS でない場合は，最初に CPS 変換をして CPS インタプリタにしておく必要がある．

今後の課題として，OCaml に algebraic effects を追加した構文を持つ Multicore OCaml に対応したステッパの実装を目指している．基本的な動作は我々が定義した言語のインタプリタと同様なので，インタプリタ関数を用意できれば変換によってステッパが導出できると考えている．

6 まとめ

ステッパを実装するためには，コンテキストの情報を保持しながら部分式を再帰的に実行するインタプリタを作ればよい．以前の研究 [3] では言語ごとにコンテキストを表すデータ型を考えた上でインタプリタに実行の流れに従った新しい引数を付け足す作業が必要だった．それに対して本研究では通常のインタプリタを CPS 変換および非関数化するという機械的な操作でコンテキストの型およびコンテキストの情報を保持するインタプリタ関数を導出した．

その方法で，継続を明示的に扱える algebraic effects を含む言語に対するステッパを実装し，他の例外処理機能である try-with や shift/reset を含む言語についても同様の変換ができることを確認した．

参考文献

[1] J. Clements, M. Flatt, and M. Felleisen. Modeling an algebraic stepper. In *ESOP 2001*, pp. 320–334. Springer, 2001.

[2] O. Danvy and A. Filinski. Abstracting control. In *LFP '90*, pp. 151–160, 1990.

[3] T. Furukawa, Y. Cong, and K. Asai. Stepping OCaml. In *TFPIE 2018*, *Electronic Proceedings in Theoretical Computer Science*, Vol. 295, pp. 17–34, 2019.

[4] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, Vol. 1, No. 2, pp. 125–159, 1975.

[5] M. Pretnar. An introduction to algebraic effects and handlers. *Electronic Notes in Theoretical Computer Science*, Vol. 319, pp. 19–35, 2015.

[6] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *HOSC*, Vol. 11, No. 4, pp. 363–397, Dec 1998.

[7] J. Whitington and T. Ridge. Direct interpretation of functional programs for debugging. In *ML 2017*, *Electronic Proceedings in Theoretical Computer Science*, Vol. 294, pp. 41–73, 2019.