OCaml ステッパの改良

秋山 雛乃　(指導教員：浅井 健一)

1　はじめに

本研究では古川ら [1] が実装した OCaml ステッパを扱う。OCaml ステッパは OCaml バージョン 4.04.0 用で、プログラムの計算過程を 1 ステップごとに出力するプログラムである。そのため、プログラミング初学者が理解のために利用するだけでなく、デバックの際にも利用されることを想定している。例えば、

```
let f x = x + 1 in 8 + (f 3)
```

というプログラムを入力すると、以下のように出力される。

```
(* Step 0 *) let f x = x + 1 in 8 + (f 3)
(* Step 1 *) 8 + ((fun x -> x + 1) 3)
(* Step 1 *) 8 + ( (fun x -> x + 1) 3 )
(* Step 2 *) 8 + ( 3 + 1 )
(* Step 2 *) 8 + ( 3 + 1 )
(* Step 3 *) 8 + ( 4 )
(* Step 3 *) 8 + 4
(* Step 4 *) 12
```

本研究では、OCaml ステッパを OCaml のバージョン 4.04.0 から 4.11.1 に対応させ、コンテキストを変更した。その後例外処理とモジュールに対応させることでステッパの改良を行った。

2　OCaml 4.11.1 に対応

ステッパを実装するために compiler-libs を用いる。compiler-libs には OCaml コンパイラを構成する各モジュールがライブラリとして提供されている。その中で構文木が出来上がった部分の後にフックをかけ、先行研究 [1] を参考にステッパを作成する。この時、OCaml のバージョンに合わせた型変更と、コンテキストの変更を行った。

2.1　OCaml のバージョンに合わせた型変更

まず OCaml のバージョンが 4.04.0 から 4.11.1 に変化したことにより、抽象構文木の型が変化した。例えば、OCaml の構文木のうち式を表現している `expression` 型は、OCaml 4.04.0 では

```
type expression =
{pexp_desc: expression_desc;
 pexp_loc: Location.t;
 pexp_attributes: attributes;}
```

だが、OCaml 4.11.1 ではバージョンが進むにつれ構文木に色々な情報が付加されたため、以下のように `pexp_loc_stack` というフィールドが増えた。

```
type expression =
{pexp_desc: expression_desc;
 pexp_loc: Location.t;
 pexp_loc_stack: location_stack;
 pexp_attributes: attributes;}
```

それにより、抽象構文木に合わせたプログラムに変更した。

2.2　コンテキストの変更

次にコンテキストの変更を行った。コンテキストとは現在実行しているプログラムの周りの状況を表すものである。例えば

```
if 1 = 1 then true else false
```

という式があったとき、まず `1 = 1` が計算される。その時、プログラムの周りの状況は `if [.] then true else false` でありコンテキストとして追加する。先行研究 [1] ではコンテキストをダイナミックバインディングにより導入していたが、本研究では明示的に引数として渡すことで直感的にコンテキストの状態を追うことが可能となった。

3　例外処理

OCaml には例外処理をする構文がある。先行研究 [1] では例外処理が含まれるプログラムを実行する手法が提案されているが、実装がうまく動いていなかったため、本研究では先行研究 [1] の方針に従って実装し直した。その時コンテキストの型に注目した。例えば、

```
try 1 + (2 + 3 / 0)
with Division_by_zero -> 0
```

という式があった時、`try-with` の内側で `Division_by_zero` という例外が起きるため、答えは `0` となる。この時 `try-with` の内側にある `1 +` や `2 +` は計算せずに `try-with` の内側を抜ける。そのため、コンテキストに追加された足し算は捨てる必要がある。これをコンテキストをリストのリストにすることで実装する。変更前はリストとなっていて、上記の例の `3 / 0` を計算するとき、コンテキストは

```
[2 + [.]; 1 + [.];
 try [.] with Division_by_zero -> 0]
```

である。変更後は以下のように直近の `try-with` までのコンテキストを明示化し、捨てることが可能となった。

```
[[2 + [.]; 1 + [.]];
 [try [.] with Division_by_zero -> 0]]
```

4　モジュール

OCaml のモジュールとは、五十嵐 [2] によると定義の集合に名前をつけ、その定義の詳細を抽象化・隠蔽する機能をもつ。さらにモジュールを別のファイルに宣言することや、モジュール上の関数であるファンクタを用いることができる。先行研究 [1] では List モジュールなどを限定的に実装していたが、本研究では同ファイル内のモジュール宣言一般を実装する。

4.1　ストラクチャと attribute

OCaml のプログラムは let 文やモジュール宣言などのストラクチャアイテムのリストであるストラクチャによって作られている。そのため、先行研究 [1] では各ストラクチャアイテムに対して再帰的に計算をしていく。そして let 文が出てきた時には、それ以降のストラクチャに対して変数の中身を代入するため同じ変数があるか調べる。もし同じ変数が出てきた場合はすぐに代入するのではなく、その変数を使う時に中身を代

入できるように attribute として情報を付与する。また、同じ変数名の定義が出てきた場合は、それ以降のストラクチャに対して attribute を付与しない。例えば以下のようなプログラムを実行する時、

```
let x = 5
let y = x
let x = 9
let z = x
```

まず、変数 x に 5 を代入するという attribute を 2 行目の x に付与する。その後 3 行目で x の定義がされているので 4 行目の x には attribute が付与されない。そして、2 行目が実行される時、付与された attribute を参考に x に 5 が代入される。

4.2 実装

本研究では、モジュールを OCaml ステッパに実装するため二つの変更を行った。変数の代入をモジュール宣言の内側にも行う点と、モジュール宣言の内側と外側のストラクチャを区別する点である。4.3 節と 4.4 節では以下のモジュール宣言の例を用いる。

```
let a = 1 + 2
module M = struct let test1 = a
                  let a = 8
                  let test2 = a end
let test3 = a + M.a
```

このプログラムは以下のようなステップ実行となる。

```
(* Step 1 *)
let a = (3)
```

まず一番上の a の足し算を行う。この時、test1 と test3 の a に対して a に 3 を代入するという attribute を付与する。

```
(* Step 2 *)
let a = 3
let test1 = (3)
```

次に M モジュールの内側に入り、test1 の a に 3 が代入される。

```
(* Step 3 *)
let a = 3
let test1 = 3
let a = 8
let test2 = (8)
```

その後、a の変数定義が新たに行われるため、test2 の a に対して 8 を代入するという attribute が付与され、代入を行う。

```
(* Step 4 *)
let a = 3
module M = struct let test1 = 3
                  let a = 8
                  let test2 = 8 end
let test3 = a + (8)
```

そして M モジュールを抜けるとモジュールの内側のストラクチャがモジュール宣言の内側に格納される。この時、モジュール宣言の内側のストラクチャの let 文をモジュール宣言の外側に代入するために attribute が付与される。よって test3 の M.a に M モジュール内の変数 a の中身である 8 が代入されるという attribute が付与され、その後代入を行う。

```
(* Step 5 *)
let a = 3
module M = struct let test1 = 3
                  let a = 8
                  let test2 = 8 end
let test3 = (3) + 8
```

次に test3 の a に 3 が代入され、

```
(* Step 6 *)
let a = 3
module M = struct let test1 = 3
                  let a = 8
                  let test2 = 8 end
let test3 = (11)
```

最後に足し算が行われる。

4.3 変数の代入

まず変数の代入をモジュール宣言の内側にも行うように変更した。4.2 節の例のステップ実行の step1 の部分では、test1 や test3 で出現する a という変数に 3 を代入するという attribute を付与する必要がある。しかし、先行研究 [1] ではモジュールの内側のストラクチャに対して attribute を付与することができず、test1 の a に 3 を代入することができなかった。そのため、モジュール宣言の外側だけでなく内側にも attribute を付与できるようにプログラムを変更した。この時、4.2 節の例での M モジュール内の変数名 a の定義のように、モジュール宣言の内側に同名の変数定義があった場合、attribute を付与する計算をやめてしまうが、モジュールの外側に出た後に出現する変数にはきちんと attribute がつくように工夫した。

4.4 ストラクチャを区別

次に、モジュール宣言の内側と外側のストラクチャを区別するように変更した。モジュール宣言のストラクチャアイテムの中にはモジュール宣言の内側のストラクチャが入っており、モジュール宣言の内側も再帰的に計算される。4.2 節の例のステップ実行では step2 と step3 がモジュール内の計算に該当している。その後モジュール宣言の内側のストラクチャの計算が終わるとストラクチャの中身をモジュール宣言のストラクチャアイテムの中に入れ直す作業を行う。そのため、step4 では M モジュールが宣言されたプログラムとなっている。この時、今まで計算してきたストラクチャの中でどの範囲がモジュール宣言の内側のストラクチャなのか知るために、モジュール宣言の内側と外側のストラクチャを区別する必要がある。そのため、計算し終わったストラクチャをモジュール宣言の内側か外側かに分けて格納するデータ構造を実装した。

5 まとめと今後の課題

OCaml ステッパを最新のバージョンとより多くの構文に対応させることで、ステッパの使いやすさを向上した。また、モジュールが含まれるプログラムをどうステップ実行すれば良いかを検討し、明らかではなかったステップ実行法を見出した。今後はモジュール宣言の表示改善や別ファイルのモジュール宣言などを実装しより良いステッパを目指したい。

参考文献

[1] Furukawa, T., Cong, Y. and Asai, K.: Stepping OCaml, *Electronic Proceedings in Theoretical Computer Science*, Vol. 295, p. 17–34 (2019).

[2] 五十嵐淳：プログラミング in OCaml, 技術評論社 (2007).