shift/reset を含む部分評価器の実装

対馬かなえ　(指導教官:浅井健一)

1　はじめに

本研究では [2, 3] を参考にして部分評価器を実装し、それを [1] のように shift/reset を含むように拡張することを目的とする。それによって shift/reset を含むプログラムの部分評価が可能となり、実行時により速く計算出来るようになることが期待される。

2　継続、shift/reset とは

継続とは式の評価時にその後にする仕事のことである。shift/reset とは継続を限定して扱うための命令である。shift は限定された継続を切り取ってくる命令、reset は shift の切り取る継続を限定する命令である。

例えば `1 + reset (2 * (shift k → k 3))` という式を考えると、`shift` が切り取ってくる継続は `reset` の内側に限定された `2` を掛けるという部分のみであり、`reset` の外側の `1` を足すという部分は含まない。この `2` を掛けるという部分は `(fun x → 2 * x)` という関数に相当する。よって `k 3` は `(fun x → 2 * x) 3` となり、`6` と評価される。次に `2 *` の部分は `shift` によって切り取られてしまったため、計算されず、最後に `1 +` を評価して最終結果は `7` となる。

3　部分評価

部分評価とは、主に計算時間の短縮を目的としてプログラム中の評価出来る「部分」を「評価」したプログラムを作る処理である。プログラムを見ながら部分評価していくオンライン部分評価と、プログラムを評価可能な所と評価不可能な所に分け (束縛時解析) それから評価 (特化) していくオフライン部分評価があるが、本研究ではオフライン部分評価を扱う。

例として x の y 乗を求める関数

```
let rec power x y =
  if y = 0 then 1 else x * (power x (y - 1))
```

を `(fun m → (power m 2))` と呼び出す場合を考える。`m` の値が不明であるが、評価 (関数展開) していくと、`(fun m → m * m * 1)` という 2 乗に特化した元のプログラムと全く同じ計算をする関数が得られる。このように評価出来る所を出来るだけ評価していくというのが部分評価である。

4　構文

今回扱う構文は OCaml を簡略化したもので、束縛時解析後は構文が Static/Dynamic の 2 レベルとなる。

5　束縛時解析

束縛時解析ではプログラム中の式を評価可能 (Static) な部分と評価不可能 (Dynamic) な部分に分けていく。評価可能とは計算して値にすることが出来るということであり、評価不可能とは式として残す必要があるということである。Static な式は通常と同じ構文で書き、Dynamic な式はその前に "_" を付けることで表す。

例えば `(fun m → m + (1 + 2))` では、`(1 + 2)` は評価出来るが、`m` の値は分からないので `m + (1 + 2)` を評価することは出来ない。従って `(1 + 2)` は Static、`m` は Dynamic、式全体は Dynamic となる。

加算の部分を型規則で表すと以下のようになる。

$$\frac{\Gamma \vdash \mathtt{E}_1 : i_1 \quad \Gamma \vdash \mathtt{E}_2 : i_2}{\Gamma \vdash \mathtt{E}_1 + \mathtt{E}_2 : i} \qquad i_2 \to i,\ i_1 \to i$$

$i_2 \to i$ は i_2 が Dynamic であるならば i も Dynamic になることを表す。これは加算の左右の項の片方でも Dynamic になったら、加算全体が Dynamic になることを示している。

Dynamic な式の中に Static な式が発生する時には、Static な式を Dynamic な式として扱うための `lift` という式を用いる。これを用いて上の例を束縛時解析すると `(fun m → m _+ lift(1 + 2))` となる。`m` に足すという式は評価出来ないが、`1 + 2` は評価出来ると束縛時解析されているのが分かる。この式の特化を行うと `(fun m → m + 3)` となる。

`lift` の式の中に入り得るのは数字かブール値に評価出来る Static な式のみであり、関数は入り得ない。

5.1　polyvariant な束縛時解析

ここで以下のプログラムを考える。

```
let rec power x y =
  if y = 0 then 1 else x * (power x (y - 1))
in (fun m → (power m 2 + power 2 m))
```

この関数 `power` は、`power m 2` では `x` が Dynamic となり、`power 2 m` では `y` が Dynamic となる。従って両者で一つの関数を共有すると、引数全てが Dynamic となり、3 章で部分評価可能だった `power m 2` も部分評価不可能になってしまう。そこで引数の Static/Dynamic に応じて関数を新たに作り、それぞれを束縛時解析していくことにする。

上のプログラムは

```
let rec power1 x y =
  if y = 0 then (lift 1)
  else x _* (power1 x (y - 1)) in
```

```
let rec power2 x y =
  _if y _= (lift 0) then (lift 1)
  else
    (lift x) _* (_power2 x (y _- (lift 1)))
in (_fun m → (power1 m 2 _+ _power2 2 m))
```

のように power1 は power の引数が Dynamic、Static な場合、power2 は Static、Dynamic な場合とそれぞれ束縛時解析される。

5.2 shift/reset を含む束縛時解析

継続が含まれると、型は式の型に加えて答えの前後の型が必要になる。前述の型規則は以下のようになる。

$$\frac{\Gamma,\ t_2 \vdash \mathtt{E}_1 : i_1,\ t_1 \quad \Gamma,\ t_3 \vdash \mathtt{E}_2 : i_2,\ t_2}{\Gamma,\ t_3 \vdash \mathtt{E}_1 + \mathtt{E}_2 : i,\ t_1} \qquad i_2 \to i,\ i_1 \to i$$

これは t を無視して、i のみ見れば shift/reset を含まない型規則と同一である。加えて下の shift/reset の構文の型規則が追加される。shift の型規則は、

$$\frac{\Gamma,\ \mathtt{x} : \mathtt{Fun}(t_2,\ t_3,\ [t_6],\ t_1,\ t_3),\ t_4 \vdash \mathtt{E} : t_4,\ t_5}{\Gamma,\ t_1 \vdash \mathtt{shift\ x} \to \mathtt{E} : t_2,\ t_5}$$

$t_6 \to t_2,\ t_6 \to t_3,\ t_6 \to t_1$

となる。t_6 とは Fun 全体の型であり、t_6 が Static ならばその Fun の型を持つ式が評価可能であることを示している。t_6 が Dynamic になれば、引数も含めて Fun 全体が Dynamic になる。

また、reset の型規則は以下のようになる。

$$\frac{\Gamma,\ t_1 \vdash \mathtt{E} : t_1,\ t_2}{\Gamma,\ t_3 \vdash \mathtt{reset\ E} : t_4,\ t_3}$$

if t_2 = Dynamic/Fun then $t_4 = t_2$

shift/reset は式の型を増やし、上の規則を加えるだけで実装出来ている。

6 特化

特化とは評価可能な式を評価し、評価不可能とされた式も与えられた値専用のプログラムが作れるならば作り、より評価を進めることである。それぞれ考察すべき状況とその例があるので、それを以下で示す。

6.1 再帰変数

再帰を含む関数の場合、関数展開してよいかどうかは、再帰に関わる変数 (再帰変数) が分かっているかどうかに関係してくる。再帰変数が分かっていると、無限の関数展開が避けられるため関数展開が行え、より特化が出来る。しかし、

```
let rec power x y =
   if y = 0 then 1 else x * (power x (y - 1))
in (fun m → power 2 m)
```

というプログラムを考えると、再帰変数となっている m の値が分からないので y = 0 かどうかはいつまでたっても決定出来ず、無限に展開されてしまう。

関数展開可能か否かの判断は一般には決定不能なので、ユーザがプログラム入力時に関数の引数が再帰関数か否かを、引数の後にブール値で指定するものとする。

6.2 特化

束縛時解析の結果が Dynamic なら評価不可能である。しかし 3 章のプログラムの場合、一つ目の引数が 2 であることが分かっているので x を 2 で置き換えて

```
let rec power2 y =
   if y = 0 then 1 else 2 * (power2 (y - 1))
in (fun m → power2 m)
```

とすることが可能である。引数を減らしその場合に特化した関数を作るのでこの処理は特化という。

6.3 特化変数

しかしいつでも特化を行ってよい訳ではなく、特化を行うと無限に特化を行ってしまうような例もある。例えばアキュムレータを使った x の y 乗を求める関数

```
let rec power x:false y:true acc:false =
  if y = 0 then acc
  else (power x (y - 1) (x * acc))
```

で (fun m → power 2 m 1) と呼び出された場合に特化を行うと acc が 1 のバージョン、2、4、··· と無限に作られ、特化が止まらなくなる。

特化をしたい引数であるかどうかも関数の引数の後にユーザが指定することにする。上の関数では

```
let rec power x:false,true y:true,true
                                acc:false,false =
  if y = 0 then acc
  else (power x (y - 1) (x * acc))
```

と指定される。

7 まとめと今後の課題

本研究では polyvariant な再帰関数の束縛時解析を定式化し、shift/reset を含むプログラムの部分評価器を実装した。これによって関数の引数の Static/Dynamic に応じて関数が複製され、polyvariant な特化が行えるようになった。

今後は shift/reset を含む様々なプログラムで特化がどのように行われていくか見ていきたい。

参考文献

[1] Kenichi. Asai “Logical Relations for Call-by-value Delimited Continuations”, In *A chapter of Trends in Functional Programming, Vol. 6*, pp. 63–78, Intellect (2007).

[2] 阪上紗里 “MinCaml Compiler における関数展開の制御”, お茶の水女子大学 卒業論文 2006.

[3] 長崎玲 “オフライン部分評価器の実装”, お茶の水女子大学 卒業論文 2004.