Type-Directed Partial Evaluation in ML

すみい

1998 年 7 月 21 日

要旨

Danvy の Type-Directed Partial Evaluation [2] を ML 上で実装した。型つけの問題などを解決するために、いくつかの超絶技巧が必要である。

1 Partial Evaluation

プログラム $f(x,y)$ とその入力の一部 $x = x_0$ を受け取って、それに対して specialize されたプログラム $f_{x_0}(y)$ を生成する。 $PE(f,x_0) = f_{x_0}$ ならば $f(x_0,y) = f_{x_0}(y)$ が成り立ち、右辺はしばしば左辺よりも efficient に計算できる。普通は source-to-source の変換として実装される。

例 1 $f(x,y) = (x-1)^2 + (y-2)^2$ を $x = 3$ に対して specialize して、 $f_3(y) = 4 + (y-2)^2$ を得る。 □

例 2 `sqr(x) = x * x` として、

```
power(m, n) =
  if n = 0 then 1
  else if n mod 2 = 0 then sqr(power(m, n div 2))
  else m * (power(m, n - 1))
```

を `n = 10` に対して specialize して

```
power_10(m) = sqr(m * (sqr(sqr(m))))
```

を得る。 □

2 Scheme 上の TDPE

すでに評価されてしまった Scheme の closed な value v と、その型 τ を受け取って、評価すると v になるような expression の normal form e を構成する。

以下において $\overline{\lambda} x.\, v$ は `(lambda (x) v)` という value を、$\underline{\lambda} x.\, e$ は `‘(lambda (x) ,e)` という expression を表すものとする。同様に、 $f \mathbin{\overline{@}} x$ は `(f x)` という value を、 $f \mathbin{\underline{@}} x$ は `‘(,f ,x)` という expression を表す。

例 1 b を何らかの base type として、

$$\begin{aligned} \tau &= b \to b \to b \\ v &= \overline{\lambda} x.\, \overline{\lambda} y.\, x \end{aligned}$$

とする。もちろん Type-Directed Partial Evaluator はまだ $\underline{\lambda} x.\ \underline{\lambda} y.\ x$ という expression を知らない。しかし、 v は $b \to b \to b$ という型を持つので、 base type の value を受け取る関数であることはわかる。そこで、

unique な symbol $'a$ を生成して v に渡し、戻り値を w とする。 w は $b \to b$ という型を持つので、やはり base type の value を受け取る関数である。そこで、ふたたび unique な symbol $'b$ を生成して w に渡すと、今度は $'a$ が返ってくる。したがって、 w は $\underline{\lambda} b.\ 'a$ と表せる。 v は $\underline{\lambda} a.\ w$ と表せるから、結局 $\underline{\lambda} a.\ \underline{\lambda} b.\ 'a$ すなわち `'(lambda (a) (lambda (b) a))` という expression が得られる。 □

例 **2**

$$\begin{aligned}\tau &= (b \to b) \to b \to b \\ v &= \overline{\lambda} f.\ \overline{\lambda} x.\ (fx)\end{aligned}$$

今度は v に b ではなく $b \to b$ 型の値を渡さなければならない。そこで、やはり unique な symbol $'a$ を生成するが、 $'a$ 自体ではなく $\overline{\lambda} x.\ 'a \,\underline{@}\, x$ という $b \to b$ 型の値を作り、 v に渡す。さらに $'b$ を作って渡すと、 $'a \,\underline{@}\, 'b$ が返ってくるので、めでたく v は $\underline{\lambda} a.\ \underline{\lambda} b.\ 'a \,\underline{@}\, 'b$ つまり `'(lambda (a) (lambda (b) (a b)))` と表せる。 □

上のようにして value を expression に変換することを reification といい、逆に expression を value に変換することを reflection という。関数型の value は reify する際に apply されるので、その際に冗長な部分は簡約されてしまうため、結果として得られる expression は long β-η-normal form になる。

実装例：

```
(define id 0)

(define (gensym!)
  (set! id (1+ id))
  (string->symbol (string-append "x" (number->string id))))

(define (reify t v)
  (cond
   [(eq? (car t) 'base)
    v]
   [(eq? (car t) 'func)
    (let ([x (gensym!)])
      `(lambda (,x) ,(reify (cddr t) (v (reflect (cadr t) x)))))]
   [(eq? (car t) 'pair)
    `(cons ,(reify (cadr t) (car v)) ,(reify (cddr t) (cdr v)))]))

(define (reflect t e)
  (cond
   [(eq? (car t) 'base)
    e]
   [(eq? (car t) 'func)
    (lambda (v) (reflect (cddr t) `(,e ,(reify (cadr t) v))))]
   [(eq? (car t) 'pair)
    (cons (reflect (cadr t) `(car ,e)) (reflect (cddr t) `(cdr ,e)))]))
```

実行例 **1:**

```
> (define S
    (lambda (f) (lambda (g) (lambda (x) ((f x) (g x))))))
> (define K
    (lambda (a) (lambda (b) a)))
> (define I ((S K) K))
> (define t '(func . ((base . T) . (base . T))))
> (reify t I)
(lambda (x1) x1)
```

実行例 2:

```
> (define app
    (lambda (x) ((car x) (cdr x))))
> (define t
    '(func . ((pair . ((func . ((base . T)
                                . (base . T)))
                       . (base . T)))
              . (base . T))))
> (reify t app)
(lambda (x2) ((car x2) (cdr x2)))
```

integer などの primitive に関する演算があっても、 override して良いなら reify できる (λ-abstract して close してしまう手もある)。

実行例 3:

```
> (define (1+ x)
    (if (integer? x)
        (+ x 1)
        `(1+ ,x)))
> (define v
    (lambda (x)
      (lambda (y)
        (lambda (f)
          (f (cons (1+ x) (1+ y)))))))
> (define t0 '(func . ((pair . ((base . T) . (base . T)))
                       . (base . T))))
> (define t `(func . ((base . T)
                      . (func . (,t0
                                 . (base . T))))))
> (reify t (v 123))
(lambda (x3) (lambda (x4) (x4 (cons 124 (1+ x3)))))
```

再帰は fix point operator を primitive な定数とみなすことにより扱える。

3 ML 上の TDPE

ML において Scheme と同様に reify/reflect を実装すると、

- reify が t に応じて様々な型の値を受け取る
- reflect が t に応じて様々な型の値を返す

ためにどちらも型つけできない。これは sprintf の話 [3] と同様の問題なので、同様のテクニックで解決できる。[1] すなわち、型 τ の representation として

[1] Thanks to 浅井さん

```
datatype typ =
    Base
  | Func of typ * typ
  | Pair of typ * typ
```

などではなく、τ 型の value/expression を reify/reflect する関数を用いれば良い。具体的には、

```
datatype ('a, 'b) which =
    Reify of 'a
  | Reflect of 'b
```

として、引数が `Reify(v)` だったら `Reify(reify(v))` を返し、引数が `Reflect(e)` だったら `Reflect(reflect(e))` を返す関数で、型を represent する。

```
datatype exp =
    Var of string
  | Abs of string * exp
  | App of exp * exp
  | Pair of exp * exp
  | Fst of exp
  | Snd of exp
  | Int of int
  | Str of string
  | Bool of bool
  | Neg of exp
  | Dbl of exp
  | Sqr of exp
  | Add of exp * exp
  | Sub of exp * exp
  | Mul of exp * exp
  | Div of exp * exp
  | Mod of exp * exp
  | Eq of exp * exp
  | Less of exp * exp
  | More of exp * exp
  | If of exp * exp * exp
  | Fix of exp
  | Let of string * exp * exp
  | Case of exp * (string * exp) * (string * exp)
  | Inleft of exp
  | Inright of exp
```

```
(* representation of base types *)

fun T (Reify(v)) = Reify(v)
  | T (Reflect(e)) = Reflect(e)

(* representation of function types *)

infixr 7 -->

fun (dom --> codom) (Reify(v)) =
    Reify(let
              val x = genvar ()
          in
              Abs(x, reify codom (v (reflect dom (Var(x)))))
          end)
  | (dom --> codom) (Reflect(e)) =
    Reflect(fn x => reflect codom (App(e, reify dom x)))

(* representation of pair types *)

infix 9 **

fun (car ** cdr) (Reify(v)) =
    Reify(Pair(reify car (fst v), reify cdr (snd v)))
  | (car ** cdr) (Reflect(e)) =
    Reflect(reflect car (Fst(e)), reflect cdr (Snd(e)))
```

reify/reflect の ‘user interface’ は次のようにかける。

```
fun reify t v =
    let
        val Reify(e) = t (Reify(v))
    in
        e
    end

fun reflect t e =
    let
        val Reflect(v) = t (Reflect(e))
    in
        v
    end
```

実行例：

```
- val S = fn f => fn g => fn x => (f x) (g x);
val S = fn : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
- val K = fn a => fn b => a;
val K = fn : 'a -> 'b -> 'a
- reify (T --> T) (S K K);
val it = Abs ("x1",Var "x1") : exp
- ppn it;
(fn x1 => x1)
val it = () : unit
- fn (f, x) => f x;
val it = fn : ('a -> 'b) * 'a -> 'b
- reify ((T --> T) ** T --> T) it;
val it = Abs ("x2",App (Fst #,Snd #)) : exp
- ppn it;
(fn x2 => ((fst x2) (snd x2)))
val it = () : unit
- fn one => fn op + => fn x => fn y => fn f => f (x + one) (y + one);
val it = fn : 'a -> ('b * 'a -> 'c) -> 'b -> 'b -> ('c -> 'c -> 'd) -> 'd
- fun Add_ (Int(0), e) = e
=   | Add_ (e, Int(0)) = e
=   | Add_ (Int(i), Int(j)) = Int(i + j)
=   | Add_ (e, f) = Add(e, f);
val Add_ = fn : exp * exp -> exp
- reify
= (T --> (T --> T --> T) --> T)
= (it (Int(1)) Add_ (Int(123)));
val it = Abs ("x3",Abs ("x4",App #)) : exp
- ppn it;
(fn x3 => (fn x4 => ((x4 124) (x3 + 1))))
val it = () : unit
-
```

4 副作用のある関数・disjoint sum type

副作用のある関数は、TDPE 時に apply してはいけないし、呼び出しの順序や回数も保存しなければならない。そこで let-insertion と呼ばれるテクニックを用いる。

実行例 1:

```
- val v = fn f => fn x => (fn a => (a, a)) (f x);
val v = fn : ('a -> 'b) -> 'a -> 'b * 'b
- val e1 = reify ((T --> T) --> T --> T ** T) v;
val e1 = Abs ("x5",Abs ("x6",Pair #)) : exp
- ppn e1;
(fn x5 => (fn x6 => ((x5 x6), (x5 x6))))
val it = () : unit
- val e2 = reify ((T -!> T) --> T --> T ** T) v;
val e2 = Abs ("x7",Abs ("x8",Let #)) : exp
- ppn e2;
(fn x7 => (fn x8 => (let val x9 = (x7 x8) in (x9, x9) end)))
val it = () : unit
```

実行例 2:

```
- val v = fn zero => fn op * => fn f => fn x => (f x) * zero;
val v = fn : 'a -> ('b * 'a -> 'c) -> ('d -> 'b) -> 'd -> 'c
- fun Mul_ (Int(0), e) = Int(0)
=   | Mul_ (e, Int(0)) = Int(0)
=   | Mul_ (Int(1), e) = e
=   | Mul_ (e, Int(1)) = e
=   | Mul_ (Int(i), Int(j)) = Int(i * j)
=   | Mul_ (e, f) = Mul(e, f);
val Mul_ = fn : exp * exp -> exp
- val e1 = reify ((T --> T) --> T --> T) (v (Int(0)) Mul_);
val e1 = Abs ("x10",Abs ("x11",Int #)) : exp
- ppn e1;
(fn x10 => (fn x11 => 0))
val it = () : unit
- val e2 = reify ((T -!> T) --> T --> T) (v (Int(0)) Mul_);
val e2 = Abs ("x12",Abs ("x13",Let #)) : exp
- ppn e2;
(fn x12 => (fn x13 => (let val x14 = (x12 x13) in 0 end)))
val it = () : unit
```

disjoint sum type の reflect も、同様に 'case-insertion' が必要になる。

実行例 3:

```
- val v =
=     fn f => fn g => fn x =>
=     (case x of
=          Left(xl) => Left(f xl)
=        | Right(xr) => Right(g xr));
val v = fn : ('a -> 'b) -> ('c -> 'd) -> ('a,'c) Sum -> ('b,'d) Sum
- val e = reify ((T --> T) --> (T --> T) --> T ++ T --> T ++ T) v;
val e = Abs ("x15",Abs ("x16",Abs #)) : exp
- ppn e;
(fn x15 =>
 (fn x16 =>
  (fn x17 =>
   (case x17 of
        Left x18 =>
            (Left (x15 x18))
      | Right x19 =>
            (Right (x16 x19))))))
val it = () : unit
```

-!> や ++ はどのように定義すれば良いか？ let は

1. 副作用のある関数が呼び出されようとしたときに
2. その context においてもっとも内側にある「壁」(λ-abstraction など) の直後に

insert しなければならない。

ここで Danvy [1] は、 first-class partial continuation のための、 shift/reset [4] という primitive を用いている。普通の ML や Scheme にそんなものはないので、最初は何とかして他の方法で済まそうとしたのだが、 (不可能とはいえないが) 非常に困難だったので断念して、普通の callcc を用いた shift/reset の実装 [5] を利用することにした。[2] ただ、 callcc は純粋な SML の仕様にない (らしい) ので、これは SML/NJ に依存する。

[2] let-insertion は callcc や shift/reset を使わずに何とかできたのだが、 case-insertion はどうもうまいやり方が見つからず、約二週間に渡ってさんざん悩んだ挙げ句の果てに、何度も見たはずの [2] から [5] が refer されているのを発見して、激しく脱力した。

```
(* representation of procedure types *)

infixr 7 -!>

fun (dom -!> codom) (Reify(v)) =
    let
        val x = genvar ()
    in
        Reify(Abs(x, reset (fn () => reify codom (v (reflect dom (Var(x)))))))
    end
  | (dom -!> codom) (Reflect(e)) =
    Reflect(fn x =>
            let
                val y = genvar ()
            in
                shift (fn k =>
                       Let(y, App(e, reify dom x),
                           reset (fn () =>
                                  k (reflect codom (Var(y))))))
            end)
```

5 簡単な応用

単純な手続き型言語 (Comm とする) の interpreter を ML で書き、いわゆる first Futamura-projection により、 Comm のプログラムを ML のプログラムに変換してみた。

準備

```
datatype (''var, 'value) comm =
    Imm of 'value * ''var
  | Add of ''var * ''var * ''var
  | Seq of (''var, 'value) comm * (''var, 'value) comm
  | Cond of ''var * (''var, 'value) comm * (''var, 'value) comm
  | While of ''var * (''var, 'value) comm

fun lookup ((y, v) :: env, x) =
    if x = y
        then v
    else lookup (env, x)

fun update (env, (x, v)) = (x, v) :: env

val ini = []
```

interpreter

実行例

```
fun exec0 prog =
    fn op + => fn iszero => fn IF => fn fix =>
    fn lookup => fn update =>
    let
        fun exec (Imm(v, x)) env =
            update (env, (x, v))
          | exec (Add(x, y, z)) env =
            update (env, (z, (lookup (env, x)) + (lookup (env, y))))
          | exec (Seq(c, d)) env =
            exec d (exec c env)
          | exec (Cond(x, ct, ce)) env =
            IF (iszero (lookup (env, x)),
                fn () => exec ce env,
                fn () => exec ct env)
          | exec (While(x, c)) env =
            let
                val loop =
                    fix (fn iter =>
                         fn env =>
                         IF (iszero (lookup (env, x)),
                             fn () => env,
                             fn () => iter (exec c env)))
            in
                loop env
            end
    in
        exec prog
    end
```

compiler

```
fun toexp (Imm(i, x)) = Imm(Int(i), Str(x))
  | toexp (Add(x, y, z)) = Add(Str(x), Str(y), Str(z))
  | toexp (Seq(c, d)) = Seq(toexp c, toexp d)
  | toexp (Cond(x, c, d)) = Cond(Str(x), toexp c, toexp d)
  | toexp (While(x, c)) = While(Str(x), toexp c)

fun dexec prog =
    exec0
    (toexp prog)
    Add_
    (fn e => Eq_ (e, Int(0)))
    (If_ T)
    (Fix__ (T -!> T) const)

fun compile prog =
    reify
    ((T ** T -!> T) --> (T ** (T ** T) -!> T) --> T --> T)
    (dexec prog)
```

実行例

```
- val prog =
=     Seq(Seq(Imm(0, "s"),
=             Imm(~1, "d")),
=         While("i",
=               Seq(Add("s", "i", "s"),
=                   Add("i", "d", "i"))));
val prog = Seq (Seq (Imm #,Imm #),While ("i",Seq #)) : (string,int) comm
- val e = compile prog;
val e = Abs ("x1",Abs ("x2",Abs #)) : exp
- ppn e;
(fn x1 => (fn x2 => (fn x3 => (let val x4 = (x2 (x3, ("s", 0))) in
(let val x5 = (x2 (x4, ("d", ~1))) in (let val x16 = ((fix (fn x6 =>
(fn x7 => (let val x8 = (x1 (x7, "i")) in (if (x8 = 0) then x7 else
(let val x9 = (x1 (x7, "s")) in (let val x10 = (x1 (x7, "i")) in (let
val x11 = (x2 (x7, ("s", (x9 + x10)))) in (let val x12 = (x1 (x11,
"i")) in (let val x13 = (x1 (x11, "d")) in (let val x14 = (x2 (x11,
("i", (x12 + x13)))) in (let val x15 = (x6 x14) in x15 end) end) end)
end) end) end) end)) end)))) x5) in x16 end) end) end))))
val it = () : unit
```

上の e を (手動で) 変数を rename してきれいに書き直すと、次のようになる。このような renaming をある程度自動で行う方法も Danvy [1] が提案している。

```
fn lookup =>
fn update =>
fn env0 =>
let
    val env1 = (update (env0, ("s", 0)))
    val env2 = (update (env1, ("d", ~1)))
in
    fix
    (fn iter =>
     fn e0 =>
     let
         val i0 = lookup (e0, "i")
     in
         if i0 = 0
             then e0
         else
             let
                 val s0 = lookup (e0, "s")
                 val i1 = lookup (e0, "i")
                 val e1 = update (e0, ("s", s0 + i1))
                 val i2 = lookup (e1, "i")
                 val d0 = lookup (e1, "d")
                 val e2 = update (e1, ("i", i2 + d0))
             in
                 iter e2
             end
     end)
    env2
end
```

6 おわりに

TIC'98 で発表されていた Danvy の実装が見たかったので、二回ほどメールを送ったら、二回目の返事で

On a different matter, when you come to Europe in September (assuming you do), would you like to stop by BRICS and talk tdpe? We could support your local expenses.

といわれてしまったのですが、どうしたものでしょうか（笑）。

Appendix 1: sprintf

```
- fun i_ k s x = k (s ^ (Int.toString x));
val i_ = fn : (string -> 'a) -> string -> int -> 'a
- fun s_ k s x = k (s ^ x);
val s_ = fn : (string -> 'a) -> string -> string -> 'a
- fun l_ x k s = k (s ^ x);
val l_ = fn : string -> (string -> 'a) -> string -> 'a
- fun n_ k s = k (s ^ "\n");
val n_ = fn : (string -> 'a) -> string -> 'a
- fun sprintf c = c (fn s : string=> s) "";
val sprintf = fn : ((string -> string) -> string -> 'a) -> 'a
- sprintf (i_ o l_ " plus " o i_ o l_ " equals " o i_ o n_);
val it = fn : int -> int -> int -> string
- it 3 7 10;
val it = "3 plus 7 equals 10\n" : string
```

Appendix 2: shift/reset

```
(*** escape/coerce ***)

datatype void = VOID of void
fun coerce (VOID v) = coerce v
fun escape f =
    SMLofNJ.Cont.callcc
    (fn k => f (fn x => SMLofNJ.Cont.throw k x))

(*** shift/reset ***)

type ans = int

exception NoReset
val mk : (ans -> void) ref =
    ref (fn _ =>
         raise NoReset)
fun abort x = coerce (! mk x)

fun reset t =
    escape (fn k =>
            let
                val m = ! mk
            in
                (mk := (fn r => (mk := m; k r));
                 abort (t ()))
            end)
fun shift h =
    escape (fn k =>
            abort (h (fn v =>
                      reset (fn () =>
                             coerce (k v)))))
```

使用例

```
- 3 * (reset (fn () => 5 + (shift (fn k => k 7))));
val it = 36 : ans
- 3 * (reset (fn () => 5 + (shift (fn k => (k 7) * (k 10)))));
val it = 540 : ans
- 3 * (reset (fn () => 5 + (shift (fn k => 7))));
val it = 21 : ans
- shift (fn k => 123);
stdIn:166.1-166.20 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)

uncaught exception NoReset
  raised at: stdIn:149.9-149.16
```

参考文献

[1] Olivier Danvy. Pragmatic aspects of type-directed partial evaluation. Technical Report RS-96-15, Basic Research in Computer Science, 1996. Available from `http://www.brics.dk/RS/96/15/BRICS-RS-96-15.ps.gz`.

[2] Olivier Danvy. Type-directed partial evaluation. In *Proceedings of the 23rd Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.

[3] Olivier Danvy. Formatting strings in ML. Technical Report RS-98-5, Basic Research in Computer Science, 1998. Available from `http://www.brics.dk/RS/98/5/BRICS-RS-98-5.ps.gz`.

[4] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical Report 89/12, Institute of Datalogy, University of Copenhagen, 1989.

[5] Andrzej Filinski. Representing monads. In *Proceedings of the 21st Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1994.