Bridging the Gap between TDPE and SDPE

Eijiro Sumii

Department of Information Science,
Graduate School of Science,
University of Tokyo

Roadmap

(1) Naive Online SDPE

(2) Offline TDPE

(4) Cogen Approach to Online SDPE

(3) Online TDPE

Roadmap

(1) Naive Online SDPE

(2) Offline TDPE

(4) Cogen Approach to Online SDPE

(3) Online TDPE

What is Partial Evaluation?

- Partial Evaluation = Program Specialization

$$p = \lambda s.\ \lambda d.\ 1 + s + d$$

$$\downarrow s = 2$$

$$p_2 = \lambda d.\ 3 + d$$

- Partial Evaluation ≈ Strong Normalization

$$(\lambda s.\ \lambda d.\ 1 + s + d)\ @\ 2$$

$$\rightarrow\ \lambda d.\ 1 + 2 + d$$

$$\rightarrow\ \lambda d.\ 3 + d$$

Naive Syntax-Directed PE

- Represent Programs as Data

$$e \;::=\; \underline{x} \;\mid\; \underline{\lambda x}.e \;\mid\; e_1\underline{@}e_2$$

- Manipulate Them Symbolically

$$\begin{array}{lll}
PE(\underline{x}) & = \underline{x} & \\
PE(\underline{\lambda x}.e) & = \underline{\lambda y}.\ PE(e[\underline{y}/\underline{x}]) & \text{(where } \underline{y} \text{ is fresh)} \\
PE(e_1\underline{@}e_2) & = PE(e[PE(e_2)/\underline{x}]) & \text{(if } PE(e_1) = \underline{\lambda x}.e\text{)} \\
PE(e_1\underline{@}e_2) & = PE(e_1)\ \underline{@}\ PE(e_2) & \text{(otherwise)}
\end{array}$$

Implementation in ML

```
datatype exp = Var of string
             | Abs of string * exp
             | App of exp * exp
…
fun PE (Var(x)) = Var(x)
  | PE (Abs(x, e)) =
    let val y = gensym ()
    in Abs(y, PE (subst x (Var(y)) e))
    end
  | PE (App(e1, e2)) =
    let val e1' = PE e1
        val e2' = PE e2
    in (case e1' of
            Abs(x, e) => PE (subst x e2' e)
          | e => App(e, e2'))
```

Example

Partially Evaluate $p = \lambda s.\ \lambda d.\ s \ @\ d$

with Respect to $s = \lambda x.\ x$

$\approx$ Strongly Normalize

$p \ @\ s = (\lambda s.\ \lambda d.\ s \ @\ d) \ @\ (\lambda x.\ x)$

```
- let val p = Abs("s",
                  Abs("d",
                      App(Var "s",
                          Var "d")))
      val s = Abs("x", Var("x"))
  in PE (App(p, s))
  end;
> val it = Abs ("x1",Var "x1") : exp
```

Problems of Naive SDPE

- Naive SDPE is Complex
  - Includes an Interpreter
  - Requires one clause in the partial evaluator for one construct in the target language
- Naive SDPE is Inefficient
  - Incurs interpretive overheads such as:
    - syntax dispatch
    - environment manipulation

Roadmap

(1) Naive Online SDPE

(2) Offline TDPE

(4) Cogen Approach to Online SDPE

(3) Online TDPE

Type-Directed PE [Danvy 96]

- Originates in *Normalization by Evaluation* in Logic and Category Theory

$$\begin{array}{c}\mathbf{\texttt{value}}\\ \texttt{eval}\uparrow\ \downarrow\texttt{reify}\\ \mathbf{\texttt{exp}}\end{array}$$

$$\texttt{normalize} = \texttt{reify} \circ \texttt{eval}$$

- Exploit the Evaluator of the Meta Language

Example

```
- let fun p s d = s d
      fun id x = x
      val p_id = p id
  in reify (E-->E) p_id
  end;
> val it = Abs ("x1",Var "x1") : exp
```

How to Reify?

— When the Domain is a Base Type —

- $\downarrow_{\alpha\to\alpha} v = \underline{\lambda x}.\ v\ @\ \underline{x}$

  e.g. $\downarrow_{\alpha\to\alpha} (\lambda x.\ (\lambda y.\ y)\ @\ x)$

  $= \underline{\lambda z}.\ (\lambda x.\ (\lambda y.\ y)\ @\ x)\ @\ \underline{z}$

  $= \underline{\lambda z}.\ (\lambda y.\ y)\ @\ \underline{z}$

  $= \underline{\lambda z}.\ \underline{z}$

- $\downarrow_{\alpha\to\tau} v = \underline{\lambda x}.\ \downarrow_{\tau} (v\ @\ \underline{x})$

  e.g. $\downarrow_{\alpha\to\alpha\to\alpha} (\lambda x.\ \lambda y.\ x)$

  $= \underline{\lambda p}.\ \downarrow_{\alpha\to\alpha} ((\lambda x.\ \lambda y.\ x)\ @\ \underline{p})$

  $= \underline{\lambda p}.\ \underline{\lambda q}.\ (\lambda x.\ \lambda y.\ x)\ @\ \underline{p}\ @\ \underline{q}$

  $= \underline{\lambda p}.\ \underline{\lambda q}.\ \underline{q}$

In ML...

```
- let val f = fn x => (fn y => y) x
      val z = gensym ()
  in Abs(z, f (Var(z)))
  end;
> val it = Abs ("x1",Var "x1") : exp
- let val g = fn x => fn y => x
      val p = gensym ()
      val q = gensym ()
  in Abs(p, Abs(q, g (Var(p)) (Var(q))))
  end;
> val it = Abs ("x2",Abs ("x3",Var "x2")) : exp
```

How to Reify?

— When the Domain is a Function Type —

$\downarrow_{(\alpha\to\alpha)\to\alpha} v = \underline{\lambda x}.\ v\ @\ \underline{x}$ ⇐ Type Error

$\downarrow_{(\alpha\to\alpha)\to\alpha} v = \underline{\lambda x}.\ v\ @\ (\lambda y.\ \underline{x}\ \underline{@}\ y)$

e.g.

$$\begin{aligned}
&\downarrow_{(b\to b)\to b} (\lambda f.\ f\ @\ \underline{x}) \\
&= \underline{\lambda y}.\ (\lambda f.\ f\ @\ \underline{x})\ @\ (\lambda z.\ \underline{y}\ \underline{@}\ z) \\
&= \underline{\lambda y}.\ (\lambda z.\ \underline{y}\ \underline{@}\ z)\ @\ \underline{x} \\
&= \underline{\lambda y}.\ \underline{y}\ \underline{@}\ \underline{x}
\end{aligned}$$

In ML...

```
- let val h = fn f => f (Var("x"))
      val y = gensym ()
  in Abs(y, h (Var(y)))
  end;
Error: operator and operand don't agree
  operator domain: exp -> 'Z
  operand:         exp
  in expression:   h (Var y)
- let val h = fn f => f (Var("x"))
      val y = gensym ()
  in Abs(y, h (fn z => App(Var(y), z)))
  end;
> val it = Abs ("x1",App (Var "x1",Var "x")) : exp
```

How to Reify?

— In Genral —

$\downarrow : [\tau] \to \tau \to \mathsf{exp}$

$$\downarrow_{\alpha} v \quad = \; v$$

$$\downarrow_{\sigma\to\tau} v = \underline{\lambda x}.\, \downarrow_{\tau} (v \mathbin{@} \uparrow_{\sigma} \underline{x})$$

(where $\underline{x}$ is fresh)

$\uparrow : [\tau] \to \mathsf{exp} \to \tau$

$$\uparrow_{\alpha} e \quad = \; e$$

$$\uparrow_{\sigma\to\tau} e = \lambda x.\, \uparrow_{\tau} (e \mathbin{\underline{@}} \downarrow_{\sigma} x)$$

Implementation in ML (1)

- Straightforward Implementation Fails to Type-Check, Because $\downarrow$ and $\uparrow$ are Dependent Functions
- Solution: Represent a Type τ by a Pair of Functions ($\downarrow_\tau$, $\uparrow_\tau$) [Yang 98] [Rhiger 99]

```
datatype 'a typ = RR of ('a -> exp) * (exp -> 'a)

(* reify : 'a typ -> 'a -> exp *)
fun reify (RR(f, _)) v = f v
(* reflect : 'a typ -> exp -> 'a *)
fun reflect (RR(_, f)) e = f e
```

Implementation in ML (2)

```
(* E : exp typ *)
val E = RR(fn v => v, fn e => e)

(* --> : 'a typ * 'b typ -> ('a -> 'b) typ *)
infixr -->
fun (dom --> codom) =
RR(fn v =>
   let val x = gensym ()
   in Abs(x, reify codom (v (reflect dom (Var(x)))))
   end,
   fn e =>
   fn x => reflect codom (App(e, reify dom x)))
```

Example

```
- let val S = fn f =>
              fn g =>
              fn x => (f x) (g x)
      val K = fn a => fn b => a
      val I = S K K
  in reify (E-->E) I
  end;
> val it = Abs ("x1",Var "x1") : exp
```

More Examples

- We can use constructs of ML (but cannot residualize them)

```
- reify (E-->E-->E)
  (fn x => fn y => if 3+5<7 then x else y);
> val it = Abs ("x1",Abs ("x2",Var "x2")) : exp
```

- We may specify a non-principal type (but get a redundant result)

```
- reify ((E-->E)-->(E-->E)) (fn x => x);
> val it =
  Abs ("x3",Abs ("x4",
                App (Var "x3",Var "x4"))) : exp
```

Extensions (1): Pair Types

$e \;\; ::= \;\; \ldots \;\; | \;\; \underline{\mathrm{pair}}(e_1, e_2) \;\; | \;\; \underline{\mathrm{fst}}\ e \;\; | \;\; \underline{\mathrm{snd}}\ e$

$$\downarrow_{\sigma\times\tau} v \;=\; \underline{\mathrm{pair}}\ (\downarrow_\sigma \mathrm{fst}\ v, \downarrow_\tau \mathrm{snd}\ v)$$

$$\uparrow_{\sigma\times\tau} e \;=\; \mathrm{pair}\ (\uparrow_\sigma \underline{\mathrm{fst}}\ e, \uparrow_\tau \underline{\mathrm{snd}}\ e)$$

Extensions (2): Variant Types

$e ::= \ldots \mid \underline{\text{true}} \mid \underline{\text{false}} \mid \underline{\text{if}}\ e_0\ \underline{\text{then}}\ e_1\ \underline{\text{else}}\ e_2$

$\downarrow_{\text{bool}} v = \text{if } v \text{ then } \underline{\text{true}} \text{ else } \underline{\text{false}}$

$\uparrow_{\text{bool}} e = \text{???}$

— Want to return both "true" and "false" to the context and use the results

⇒ Manipulate *partial continuation* with "shift" & "reset" [Danvy & Finlinski 90]

Problems

- Reflection for variant types causes code duplication

$\downarrow_{(\alpha\to\alpha)\to\text{bool}\to\alpha}$
$(\lambda f.\ \lambda x.\ f\ @\ (f\ @\ (f\ @\ (\text{if } x \text{ then } \underline{y} \text{ else } \underline{z}))))$
$= \underline{\lambda f}.\ \underline{\lambda x}.\ \underline{\text{if}}\ \underline{x}\ \underline{\text{then}}\ \underline{f}\ \underline{@}\ (\underline{f}\ \underline{@}\ (\underline{f}\ \underline{@}\ (\underline{y})))$
$\qquad \underline{\text{else}}\ \underline{f}\ \underline{@}\ (\underline{f}\ \underline{@}\ (\underline{f}\ \underline{@}\ (\underline{z})))$

- Reflection for primitive/inductive types is impossible

$\downarrow_{\text{int}\to\text{int}} (\lambda n.\ 1 + 2 + n) = ???$

$\downarrow_{\text{int\_list}\to\text{int\_list}} (\lambda a.\ (\text{tl}\ (\text{tl}\ (3 :: a)))) = ???$

Roadmap

(1) Naive Online SDPE

(2) Offline TDPE

(4) Cogen Approach to Online SDPE

(3) Online TDPE

Online TDPE (1)

- Extend some primitive operators to treat residual code [Danvy 97]

  $x +' y = x + y$ (if x and y are integers)

  $x +' y = \downarrow_{\mathrm{int}} x \;\underline{+}\; \downarrow_{\mathrm{int}} y$ (if x or y is residual code)

For example:

$$\downarrow_{\mathrm{int}\to\mathrm{int}} (\lambda n.\ 1 +' 2 +' n)$$
$$= \underline{\lambda x}.\ (\lambda n.\ 1 +' 2 +' n) \mathbin{@} \underline{x}$$
$$= \underline{\lambda x}.\ 1 +' 2 +' \underline{x}$$
$$= \underline{\lambda x}.\ 3 +' \underline{x}$$
$$= \lambda x.\ 3 + x$$

In ML...

```
datatype 'a tlv = S of 'a | D of exp
val I = (* int tlv typ *)
    RR(fn D(e) => e
        | S(i) => Int(i),
       fn e => D(e))
fun add' (S(i), S(j)) = S(i + j)
  | add' (x, y) = D(Add(reify I x, reify I y))

- reify (I-->I)
  (fn n => add'(add'(S(1), S(2)), n));
> val it = Abs ("x1",Add (Int 3,Var "x1")) : exp
```

Online TDPE (2)

- Extend any value destructors to treat residual code [Sumii & Kobayashi 99]

  tl' x = tl x   (if x is a list)

  tl' x = $\underline{\text{tl}}$ x   (if x is residual code)

For example:

$$\downarrow_{int\_list \to int\_list} (\lambda a.\ (tl'\ (tl'\ (3 :: a))))$$

$$= \underline{\lambda x}.\ (\lambda a.\ (tl'\ (tl'\ (3 :: a))))\ @\ \underline{x}$$

$$= \underline{\lambda x}.\ tl'\ (tl'\ (3 :: \underline{x}))$$

$$= \underline{\lambda x}.\ tl'\ \underline{x}$$

$$= \lambda x.\ tl\ x$$

In ML...

```
datatype 'a list = nil | :: of 'a * 'a list tlv
fun L t = (* 'a typ -> 'a list tlv typ *)
    RR(fn D(e) => e
        | S(nil) => Nil
        | S(x :: y) => Cons(reify t x,
                             reify (L t) y),
       fn e => D(e))
fun tl' (D(e)) = D(Tl(e))
  | tl' (S(_ :: x)) = x

- reify (L I --> L I)
  (fn a => tl' (tl' (S(S(3) :: a))));
> val it = Abs ("x1",Tl (Var "x1")) : exp
```

Online TDPE (3)

- Extend all value destructors to treat residual code [Sumii & Kobayashi 99]

$$f \mathbin{@'} x{:}\tau = f \mathbin{@} x \qquad \text{(if f is a function)}$$

$$f \mathbin{@'} x{:}\tau = f \mathbin{\underline{@}} \downarrow_\tau x \qquad \text{(if f is residual code)}$$

$\Rightarrow$ Reflection becomes unnecessary!

An Experiment

Specialized & executed an interpreter for a simple imperative language with a tiny program (by SML/NJ 110.0.3 on UltraSPARC 168 MHz with 1.2 GB Main Memory)

	spec	exec	
(No PE)		0.30	
[Danvy 96][(*1)]	0.57	0.14	
[Danvy 97][(*2)]	0.24	0.13	
[Sumii 99][(*2)]	0.10	0.14	(msec)

(*1) abstracted out all primitive operators

(*2) removed unnecessary 's by monovariant BTA

Roadmap

(1) Naive Online SDPE

(2) Offline TDPE

(4) Cogen Approach to Online SDPE

(3) Online TDPE

Cogen Approach to Online SDPE

We are going to:

- Realize a simple & efficient online SDPE by using:
  - Higher-Order Abstract Syntax
  - Deforestation
- See a similarity between the SDPE and our online TDPE

Higher-Order Abstract Syntax

Represent binding in the target language by binding in the meta langauge

```
datatype hexp = HAbs of hexp -> hexp
              | HApp of hexp * hexp
              | HSym of string
```

For example,

```
HAbs(fn x => HApp(HAbs(fn y => y), x)) : hexp
```

represents $\underline{\lambda x}.\ (\underline{\lambda y}.\ \underline{y})\ \underline{@}\ \underline{x}$

Converter from HOAS to FOAS

```
fun conv (HAbs(f)) =
    let val x = gensym ()
    in Abs(x, conv (f (HSym(x))))
    end
  | conv (HApp(e1, e2)) = App(conv e1, conv e2)
  | conv (HSym(s)) = Var(s)
```

Online SDPE in HOAS

```
fun PE (HAbs(f)) = HAbs(fn x => PE (f x))
  | PE (HApp(e1, e2)) =
    let val e1' = PE e1
        val e2' = PE e2
    in (case e1' of HAbs(f) => f e2'
                  | _ => HApp(e1', e2'))
    end
  | PE (HSym(s)) = HSym(s)

- let val e = HAbs(fn x =>
                  HApp(HAbs(fn y => y),
                       x))
  in conv (PE e)
  end;
> val it = Abs ("x1",Var "x1") : exp
```

Deforestation (1)

A priori compose **`HAbs`**, **`HApp`** & **`HSym`** with **`PE`**

(Instead of first constructing an **`hexp`** by **`HAbs`**, **`HApp`** & **`HSym`** and then destructing it by **`PE`**)

```
fun habs'(f) = HAbs(fn x => f x)
fun happ'(e1, e2) =
    let val e1' = e1
        val e2' = e2
    in (case e1' of HAbs(f) => f e2'
                  | _ => HApp(e1', e2'))
    end
fun hsym'(s) = HSym(s)
```

Deforestation (2)

Simplify by η-reduction & inlining

```
val habs' = HAbs
fun happ'(HAbs(f), e2) = f e2
  | happ'(e1, e2) = HApp(e1, e2)
val hsym' = HSym

- conv
  (habs'(fn x => happ'(habs'(fn y => y), x)));
> val it = Abs ("x1",Var "x1") : exp
```

Comparison

Online TDPE ≈ Cogen approach to online SDPE

Reification operator

≈ Converter from HOAS to FOAS

Value destructors extended for residual code

≈ HOAS constructors composed with PE

They are more similar in dynamically-typed languages (e.g. Scheme) than in statically-typed ones (e.g. ML) [Sumii & Kobayashi 99]

Related Work

- [Helsen & Thiemann 98]

  Pointed out similarity between offline TDPE and cogen approach to offline SDPE

  (C.f. Our PE is online.)

- [Sheard 97]

  Extended Danvy's TDPE in various but *ad hoc* ways such as lazy reflection, type passing, etc.

  (C.f. Our TDPE is more simple, efficient, and powerful.)

Conclusion

- We have:
  - Reviewed Danvy's TDPE
  - Extended it with online value destructors
  - Seen the similarity of our online TDPE and cogen approach to online SDPE
- Our future work includes:
  - More integration of SDPE and TDPE
  - More sophisticated treatment of side-effects (including non-termination)