Type-Directed Partial Evaluation in Haskell [*]

Kristoffer.Rose@ENS-Lyon.fr [†]

June 10, 1998 (version 1.19)

Abstract

We implement simple standard *type-directed partial evaluation* in the pure functional programming language Haskell, using type classes.

1 Introduction

In this paper we implement *standard type-directed partial evaluation* (TDPE) of Danvy (1996) for a small subset of the pure (lazy) functional programming language Haskell (Fasel, Hudak, Peyton Jones, Wadler et al. 1992). When summarizing the definitions we rely on *combinatory reduction systems* (CRS) of Klop, van Oostrom & van Raamsdonk (1993), however, one can read the specification without knowledge of CRSs.

The document is itself a program written in "literate Haskell:" lines that begin with small numbers $_1$ $_2$ … are Haskell program lines, except we use the symbols →, ⇒, ≡, ×, ↑, and λ instead of the usual Haskell equivalents `->`, `=>`, `==`, `*`, `^`, and `\`, and let ␣ denote a blank space inside a string.

The first line of Haskell identifies the code.

```
1 module TDPE where
```

In section 2 we introduce the base "2-level" calculus that constitutes our subset language and for which we do normalization. The following section 3 shows how we implement TDPE using Haskell type classes; section 4 shows how this can be used to implement usual partial evaluation. Finally, section 5 shows several small examples and section 6 concludes.

[*]Presented at the "Normalisation by Evaluation 1998" APPSEM workshop, Chalmers, Göteborg.

[†]Address: LIP, École Normale Supérieure de Lyon, 46 Allée d'Italie, F-69364 Lyon 7, France. Home page: `http://www.ens-lyon.fr/~krisrose`.

2 2-level λ-calculus

TDPE works with 2-level λ-calculus (Nielson & Nielson 1992) as a means to study the two "binding times" of partial evaluation. TDPE uses this same idea, however, instead of seeing the levels as data annotations we see them as marks distinguishing data and code.

2.1 Terms

The types and 2-level terms used for standard TDPE are defined as follows (using higher-order abstract syntax in CRS[1] notation):

$$\alpha, \beta ::= \mathsf{b} \mid \alpha \to \beta \mid \alpha \times \beta \qquad \text{(Type)}$$

$$\mathsf{E}, \mathsf{F} ::= \mathsf{c} \mid \underline{\lambda} x.\mathsf{E}[x] \mid (\mathsf{F} \,\underline{\ \ }\, \mathsf{E}) \mid \underline{\mathsf{pair}}(\mathsf{E}_1, \mathsf{E}_2) \mid \underline{\mathsf{fst}}\,\mathsf{E} \mid \underline{\mathsf{snd}}\,\mathsf{E} \qquad \text{(Expr)}$$

$$V, W ::= \mathsf{c} \mid \overline{\lambda} z.V[z] \mid (V \,\overline{\ \ }\, W) \mid \overline{\mathsf{pair}}(V_1, V_2) \mid \overline{\mathsf{fst}}\,V \mid \overline{\mathsf{snd}}\,V \qquad \text{(Value)}$$

where b denotes a base type and c is a constant of base type.

Note how the use of higher-order abstract syntax means that we do not need a case for variables: Their meaning is clear from the use of "$\mathsf{E}[x]$" that can be read as "inside the subterm E the variable x may occur as a subterm."

When realising this in Haskell we represent types by themselves, *i.e.*, as the corresponding Haskell type, values are native Haskell values, and expressions are a data type using a higher-order function for the binding. This means that we cannot rely on the default `Show` instance to print expressions (the appendix has the one we use).

```
data Expr = LAMBDA (Expr → Expr)
          | APPLY Expr Expr

          | PAIR Expr Expr
          | FST Expr
          | SND Expr

          | BASE String
```

The last is for free variables, constants, *etc.*: all the things where we don't really care what they look like as long as they print as the right thing.

[1]Except we use non-terminal letters as meta-variables rather than the ubiquitous Z, write abstractions as _._, and meta-application as _[_].

2.2 Static Reductions

Values are reduced as follows (again in CRS notation):

$$\left(\overline{\lambda}x.V[x]\right)\overline{}\,W \to V[W] \tag{$\overline{\beta}$}$$

$$\overline{\mathrm{fst}}\left(\overline{\mathrm{pair}}(V_1,V_2)\right) \to V_1 \tag{$\overline{\mathrm{fst}}$}$$

$$\overline{\mathrm{snd}}\left(\overline{\mathrm{pair}}(V_1,V_2)\right) \to V_2 \tag{$\overline{\mathrm{snd}}$}$$

These correspond to the definitions in Haskell of function application and the standard functions for destructing pairs. So we do not have to add any Haskell to handle these.

3 Type-directed Partial Evaluation

The TDPE system itself is formulated as follows (Danvy 1996, Fig. 1); it can again be interpreted as a CRS:

$$\downarrow^{b} V \to V \tag{$\downarrow^{b}$}$$

$$\downarrow^{\alpha\to\beta} V \to \underline{\lambda}x.\downarrow^{\beta}\left(V\,\overline{}\,(\uparrow_{\alpha} x)\right) \tag{$\downarrow^{\to}$}$$

$$\downarrow^{\alpha\times\beta} V \to \underline{\mathrm{pair}}\left(\downarrow^{\alpha}(\overline{\mathrm{fst}}\,V),\downarrow^{\beta}(\overline{\mathrm{snd}}\,V)\right) \tag{$\downarrow^{\times}$}$$

$$\uparrow_{b} E \to E \tag{$\uparrow_{b}$}$$

$$\uparrow_{\alpha\to\beta} E \to \overline{\lambda}x.\uparrow_{\beta}\left(E\,\underline{}\,(\downarrow^{\alpha} V)\right) \tag{$\uparrow_{\to}$}$$

$$\uparrow_{\alpha\times\beta} E \to \overline{\mathrm{pair}}\left(\uparrow_{\alpha}(\underline{\mathrm{fst}}\,E),\uparrow_{\beta}(\underline{\mathrm{snd}}\,E)\right) \tag{$\uparrow_{\times}$}$$

The two defined function symbols, $\downarrow$ and $\uparrow$, are called *reification* and *reflection*, respectively.

The simplification proceeds by *type case analysis*. This is what Haskell offers in the form of *type class overloading*. Thus we can implement the system as a type class.

```
class ReifyReflect alpha where
 reify   ::alpha → Expr
 reflect::Expr → alpha
```

3.1 Base Types

From the viewpoint of TDPE, all “uninteresting” types are base types in the sense that values and expressions are indistinguishable because the internal structure is not known. However, we cannot define a generic “base type instance.” (This is similar to the problem encountered in the Standard ML implementations of Zhe Yang and Andrzej Filinski.) Instead we have to pretend that all base values are of type Expr. This means that we take the two rules at base type literally: the expression (or value) constructed by reification (or reflection, respectively) really *is* the base value (or expression, respectively), to convert. Thus the class instance looks as follows:

```
instance ReifyReflect Expr where
 reify   v = v
 reflect e = e
```

Furthermore, Haskell overloading requires that we treat two cases separately: *type variables*, where we can ourself decide the type, and *constants* where Haskell enforces a particular (primitive) type.

For type variables the problem is that Haskell does not permit free (native) type variables – the kind system is first order. Thus we will instead provide the user with pseudo type variables Alpha, Beta, . . . , Omega. These are just aliased to the Expr type to make the reification be the identity on types as dictated by the definition.

```
type Alpha = Expr
type Beta  = Expr
 ⋮
type Omega = Expr
```

Constants receive special treatment because we know how to convert them from values to expressions. It is an error to reflect a value of base type (we only handle “off-line” partial evaluation), unless this is an expression that already denotes a constant.

As an example constant type here is the definition for integers.

```
instance ReifyReflect Int where
 reify v = BASE (show v)
 reflect (BASE s) = read s
 reflect _        = error "Cannot␣reflect␣non-constant␣Int."
```

More are easily added, of course.

3.2 Function Types

Next the most fundamental type: function types. These are straightforward by following the definition and making the induction over types explicit:

```
instance (ReifyReflect alpha, ReifyReflect beta)⇒
         ReifyReflect (alpha→beta)
 where
  reify   v = LAMBDA (λx→reify   (apply v (reflect x)))
  reflect e = lambda (λx → reflect (APPLY e (reify   x)))
```

where the following dummy functions help emphasize the symmetry between data and code

```
lambda f = f
apply x y = x y
```

3.3 Product Types

Again the definition directly works, after the appropriate class constraints have been added.

```
instance (ReifyReflect alpha, ReifyReflect beta)⇒
         ReifyReflect (alpha,beta)
 where
  reify   v = PAIR (reify   (fst v)) (reify   (snd v))
  reflect e = pair (reflect (FST e)) (reflect (SND e))
```

where we have used the following auxiliary for symmetry.

```
pair x y = (x,y)
```

4 Partial Evaluation

Residualization is defined as follows in the original reference:

$$\text{residualize} = \text{statically-reduce} \circ \text{reify}$$

which, in our case, degenerates to the simple

```
residualize v = reify v
```

because all the static reductions are done internally by Haskell.

Other popular names for this function are also provided.

```
tdpe v = reify v
decompile v = reify v
```

Traditional partial evaluation, *i.e.*, specializing a program p to the static component s of a pair of inputs (s, d), can now be expressed using currying:

```
pe p s = residualize (curry p s)
```

5 Some examples

Even this small language is interesting. In this session we reproduce the lines of a HUGS session (marked with small dots) to illustrate this.

```
· Hugs session for:
· /usr/local/share/hugs/lib/Prelude.hs
· TDPE.lhs
```

Let us look at the source of some standard `Prelude.hs` functions. Notice that we output the result as native Haskell so it can be reentered directly into HUGS (the actual function used to print the results is given in the appendix).

```
· TDPE> residualize (fst::(Alpha,Beta) → Alpha)
· λa → fst a

· TDPE> residualize (snd::(Alpha,Beta) → Beta)
· λa → snd a

· TDPE> residualize (curry::((Pi,Xi) → Mu) → (Pi → Xi → Mu))
· λa b c → a (b,c)

· TDPE> residualize (uncurry::(Pi → Xi → Mu) → ((Pi,Xi) → Mu))
· λa b → a (fst b) (snd b)

· TDPE> residualize (id::(Alpha → Alpha))
· λa → a

· TDPE> residualize (const::(Alpha → Beta → Alpha))
· λa b → a

· TDPE> residualize ((.)::(Rho → Mu) → (Pi → Rho) → (Pi → Mu))
· λa b c → a (b c)

· TDPE> residualize (flip::(Pi → Rho → Mu) → Rho → Pi → Mu)
· λa b c → a c b
```

We can even do “real” partial evaluation with this small system. Let us illustrate this with a generic example from Danvy (1996), the simple function `bar` that takes a static and a dynamic input:

```
bar (s,d) = d (5×s)
```

This we can specialize to a value for the static input by

```
· TDPE> pe (bar::(Int,(Int → Alpha)) → Alpha) 100
· λa → a 500
```

And, finally, the canonical example: power, after suitable abstraction of the three numeric “primitives” used: `one`, `sqr`, and `×`:

```
power n one sqr (×) x = loop n where

 loop n | n ≡ 0      = one
        | odd n      = x × loop (n - 1)
        | otherwise  = sqr (loop (div n 2))
```

Partially evaluating this gives the expected result:

```
· TDPE> tdpe ((power 10)::Xi → (Xi → Xi) → (Xi → Xi → Xi) → Xi → Xi)
· λa b c d → b (c d (b (b (c d a))))
```

Or, more readably:

```
power_10 one sqr (×) x = sqr (x × (sqr (sqr (x × one))))
```

which we can use, of course, at different types.

```
· TDPE> power_10 1 (↑2) (×) 2
· 1024

· TDPE> power_10 1.0 (↑2) (×) 2.2
· 2655.99
```

6 Conclusions

We have shown how the simplest form of standard type-directed partial evaluation fits nicely with a pure functional language, and in particular how the type class mechanism of Haskell can be used to implement it elegantly: the entire source code of the system is included in this note!

Doing this, we also hope to have illustrated how easy it is to work with higher-order abstract syntax in functional languages.

We are currently extending and using this in several directions:

- We are experimenting with using the system for deforestation.
- More type forms should be included: in particular sum-types are challenging and require a significantly more complicated base CRS.
- It would be interesting to have an "on-line" version of the partial evaluator (Danvy 1998).

A Showing Terms

Nothing unusual is going on here: we merely include this to make the file self-contained and document how the terms above where printed. Really we should be using parsing and pretty-printing ...

```
instance Show Expr where
```

```
 showsPrec _ e = st e vs0 where
```

We pick our variable names from an infinite supply.

```
  vs0 = [ c:i | i ← ("":map show [1..]), c ← ['a'..'z'] ]
```

Our entry point prints a "top-level" expression.

```
  st (LAMBDA f) (v:vs)= ss"\\".ss v.sb (f (BASE v)) vs
  st e              vs = sa e vs
```

The next level takes care of outputting λ-terms as curried as possible.

```
sb (LAMBDA f) (v:vs)= ss"␣".ss v.sb (f (BASE v)) vs
sb e              vs = ss"->".sa e vs
```

Applications associate to the left.

```
sa (APPLY e1 e2) vs = sa e1 vs.ss"␣".si e2 vs
sa (FST e)       vs = ss"fst␣".si e vs
sa (SND e)       vs = ss"snd␣".si e vs
sa e             vs = si e vs
```

Finally, the innermost function ensures that anything complex is parenthesised.

```
si (PAIR e1 e2)  vs = ss"(".st e1 vs.ss",".st e2 vs.ss")"
si (BASE s)      vs = ss s
si e             vs = ss"(".st e vs.ss")"
```

All make use of the following abbreviation.

```
ss = showString
```

References

Danvy, O. (1996), Type-directed partial evaluation, *in* G. L. Steele Jr., ed., ‘Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages’, ACM Press, St. Petersburg Beach, Florida, pp. 242–257.

Danvy, O. (1998), Online type-directed partial evaluation, *in* M. Sato & Y. Toyama, eds, ‘Proceedings of the Third Fuji International Symposium on Functional and Logic Programming’, World Scientific, Kyoto, Japan. Extended version available as the technical report BRICS RS-97-53.

Fasel, J. H., Hudak, P., Peyton Jones, S., Wadler, P. et al. (1992), ‘Haskell special issue’, *SIGPLAN Notices* **27**(5).

Klop, J. W., van Oostrom, V. & van Raamsdonk, F. (1993), ‘Combinatory reduction systems: Introduction and survey’, *Theoretical Computer Science* **121**, 279–308.

Nielson, F. & Nielson, H. R. (1992), *Two-Level Functional Languages*, Vol. 34 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press.