Type-Directed Partial Evaluation in Haskell[*]

Kristoffer Høgsbro Rose
LIP,[†] Ecole Normale Supérieure de Lyon[‡]

April 29, 1998

Abstract

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

1 Introduction

Consider the following prototypical functional programming language (without any sum-types, *i.e.*, `Bool` or types made with `|`).

1.1. Definition (2-level functional programming). The "2-level" λ-terms are given by the inductive definition (or *abstract syntax*)

$$\mathrm{T} ::= B \mid \mathrm{T}_1 \to \mathrm{T}_2 \mid \mathrm{T}_1 \times \mathrm{T}_2 \tag{1}$$

$$\mathrm{V} ::= C \mid x \mid \overline{\lambda} x.\mathrm{V} \mid \mathrm{V}_0 \,\overline{\ }\, \mathrm{V}_1 \mid \overline{\mathsf{pair}}(\mathrm{V}_1, \mathrm{V}_2) \mid \overline{\mathsf{fst}}(\mathrm{V}) \mid \overline{\mathsf{snd}}(\mathrm{V}) \tag{2}$$

$$\mathrm{E} ::= C \mid x \mid \lambda x.\mathrm{E} \mid \mathrm{E}_0\ \mathrm{E}_1 \mid \mathsf{pair}(\mathrm{E}_1, \mathrm{E}_2) \mid \mathsf{fst}(\mathrm{E}) \mid \mathsf{snd}(\mathrm{E}) \tag{3}$$

where x is supposed to come from an infinite set of variables, observing Barendregt's "variable convention" (which states that names are always chosen such that capture of free variables is avoided if possible).

The reduction rules are:

$$(\overline{\lambda} x.\mathrm{V}[x]) \,\overline{\ }\, X \to \mathrm{V}[X] \tag{$\overline{\beta}$}$$

$$\overline{\mathsf{fst}}(\overline{\mathsf{pair}}(\mathrm{V}_1, \mathrm{V}_2)) \to \mathrm{V}_1 \tag{1}$$

$$\overline{\mathsf{snd}}(\overline{\mathsf{pair}}(\mathrm{V}_1, \mathrm{V}_2)) \to \mathrm{V}_2 \tag{2}$$

[*]This is a presentation of new research results, at (almost) research level.

[†]Laboratoire de l'Informatique du Parallélisme.

[‡]46, Allée d'Italie, F–69364 Lyon 07, France; ⟨Kristoffer.Rose@ens-lyon.fr⟩.

1.2. Definition (2-level η-expansion).

$$
\begin{aligned}
{\downarrow}^{B}(\mathrm{V}) &\rightarrow \mathrm{V} && ({\downarrow}^{B})\\
{\downarrow}^{\mathrm{T}_1\rightarrow\mathrm{T}_2}(\mathrm{V}) &\rightarrow \lambda x.{\downarrow}^{\mathrm{T}_2}(\mathrm{V}\,\overline{\ }\,({\uparrow}_{\mathrm{T}_1}(x))) && ({\downarrow}^{\rightarrow})\\
{\downarrow}^{\mathrm{T}_1\times\mathrm{T}_2}(\mathrm{V}) &\rightarrow \mathrm{pair}({\downarrow}^{\mathrm{T}_1}(\overline{\mathrm{fst}}(\mathrm{V})),{\downarrow}^{\mathrm{T}_2}(\overline{\mathrm{snd}}(\mathrm{V}))) && ({\downarrow}^{\times})\\
{\uparrow}_{B}(\mathrm{E}) &\rightarrow \mathrm{E} && ({\uparrow}_{B})\\
{\uparrow}_{\mathrm{T}_1\rightarrow\mathrm{T}_2}(\mathrm{E}) &\rightarrow \overline{\lambda}v.{\uparrow}_{\mathrm{T}_2}(\mathrm{E}\ ({\downarrow}^{\mathrm{T}_1}(v))) && ({\uparrow}_{\rightarrow})\\
{\uparrow}_{\mathrm{T}_1\times\mathrm{T}_2}(\mathrm{E}) &\rightarrow \overline{\mathrm{pair}}({\uparrow}_{\mathrm{T}_1}(\mathrm{fst}(\mathrm{E})),{\uparrow}_{\mathrm{T}_2}(\mathrm{snd}(\mathrm{E}))) && ({\uparrow}_{\times})
\end{aligned}
$$

This can be modeled directly in Haskell by interpreting the overlined, constructions directly as "Haskell," and the underlined as "data." This involves only one complication: coding the variables in the data part: here we merely use de Bruijn's indices.

2 Type-Directed Partial Evaluation in Haskell

We implement a Haskell module that realizes 2-level η-expansion, or (standard) *type-directed partial evaluation* (tdpe) of a simple Haskell subset.

```
module TDPE where
```

2.1 Expressions

Expressions are data values of the following obvious type.

```
data Expr = Var Vr                      -- lambdaterms
          | Lambda Vr Expr
          | Apply Expr Expr
          | Base String                 -- base values
          | Pair Expr Expr              -- product type
          | Fst Expr
          | Snd Expr
          | Nil                         -- inductive list type
          | Cons Expr Expr
```

(NB. The above definition should really be split among each case below, if we had proper literate programming available ...)

For symmetry we add the following construction functions ("λ" cannot be added as we cannot extend the syntax of Haskell):

```
apply x y = x y
pair x y = (x,y)
nil = []
cons = (:)
```

Variables are actually strings generated from their "de Bruijn level."

```
newtype Vr = Vr(String)
vr i = Vr("x"++show i)
mkVar i = Var(vr i)
```

2.2 Reification and Reflection

Two-level η-expansion is defined by two mutually recursive functions, one reifying values to expressions and the other reflecting expressions to values, corresponding to $\downarrow^{\cdot}(\cdot)$ and $\uparrow_{\cdot}(\cdot)$ of the introduction, respectively. Both take a first agument indicating the *nesting level* of the expression; this is used to create unique variable names. Furthermore, we define reification and reflection as the first and second half of one function operating on pairs to facilitate make it easy to define the default case.

A "type" is thus encoded as follows (`RR` stands for "reify-reflect pair"):

```
type Reifier t = Int → t → Expr
type Reflecter t = Int → Expr → t

newtype RR t = RR(Reifier t,Reflecter t)
```

Since the definitions of reification and reflection are type-directed we will use the Haskell *type class overloading* to define the reify-reflect pair `rr` for every type.

```
class ReifyReflect t where
 rr :: RR t
```

We can now define an instance of `ReifyReflect` for each Haskell value type that corresponds to an actual `Expr`. We start with the fundamental one for function types.

```
instance (ReifyReflect alpha,ReifyReflect beta)=>
         ReifyReflect (alpha → beta) where

 rr = RR(reif,refl) where

  reif i v = Lambda (vr i)
                (reif2 (i+1)
                       (apply v (refl1 (i+1)
                                       (Var (vr i)))))
  refl i e = λv → refl2 (i+1)
                       (Apply e (reif1 (i+1)
                                       v))

  RR(reif1,refl1) = rr :: ReifyReflect alpha=>RR alpha
  RR(reif2,refl2) = rr :: ReifyReflect beta=>RR beta
```

To permit expressing simple types we permit type variables `Alpha`, `Beta`, ..., `Omega`. These are just aliased to the `Expr` type to make the reification be the indentity on types as dictated by the definition..

```
instance ReifyReflect Expr where
 rr = RR(λi v → v,λi e → e)
```

```
type Alpha = Expr
type Beta = Expr
type Gamma = Expr
type Delta = Expr
type Epsilon = Expr
type Zeta = Expr
type Eta = Expr
type Theta = Expr
type Iota = Expr
type Kappa = Expr
type Lambda = Expr
type Mu = Expr
type Nu = Expr
type Xi = Expr
type Pi = Expr
type Rho = Expr
type Sigma = Expr
type Tau = Expr
type Upsilon = Expr
type Phi = Expr
type Chi = Expr
type Psi = Expr
type Omega = Expr
```

2.3 Base Types

"Base values" 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 "offline" partial evaluation.

The simplest base value is the unit value.

```
instance ReifyReflect () where
 rr =
  RR(λi v → Base "()",
     error "Cannot␣reflect␣base␣value␣::␣().")
```

Integers are also merely printed.

```
instance ReifyReflect Integer where
 rr =
  RR(λi v → Base (show v),
     error "Cannot␣reflect␣base␣value␣::␣Integer.")
```

2.4 Product Types

The only product type included presently is pairs, *i.e.*, tuples with two elements.

```
instance (ReifyReflect alpha,ReifyReflect beta)=>
         ReifyReflect (alpha,beta) where
 rr = RR(reif,refl) where
```

```
  reif i v = Pair (reif1 i (fst v)) (reif2 i (snd v))
  refl i e = pair (refl1 i (Fst e)) (refl2 i (Snd e))
  RR(reif1,refl1) = rr :: ReifyReflect alpha=>RR alpha
  RR(reif2,refl2) = rr :: ReifyReflect beta=>RR beta
```

2.5 Inductive Types

"Inductive types" here merely means types coded up with their *Church inductor*. We only include Church lists, corresponding to lists with a finite length (permitting induction over the length of the list).

```
type ChurchList alpha beta = (alpha → beta → beta) → beta → beta
newtype CL alpha beta = CL(ChurchList alpha beta)
l2cl :: [alpha] → CL alpha beta
l2cl l = CL (λc n → foldr c n l)
cl2l :: CL t [t] → [t]
cl2l (CL cl) = cl cons nil
nilcl :: ChurchList alpha beta
nilcl = λ(cons,nil) → nil
conscl :: (alpha,ChurchList alpha beta) → ChurchList alpha beta
conscl(x,cl) = λ(cons,nil) → cons (x, cl(cons,nil))
mapcl f (CL cl) = CL (λc n → cl (λx xs → c (f x) xs) n)
```

The fold funtional is very simple, showing the close relation between folding and the induction implicit in the Church encoding.

```
foldcl f n cl = cl(λ(x,y) → f x y,n)
```

Now we can define the reify-reflection pair for Church lists, naively.

```
instance (ReifyReflect alpha,ReifyReflect beta)=>
         ReifyReflect (CL alpha beta) where
 rr =
  RR(λi (CL v) → reif i v,
     λi e → CL (refl i e))
  where
   RR(reif,refl) = rr :: (ReifyReflect alpha,ReifyReflect beta)=>
                         RR(ChurchList alpha beta)
```

2.6 Recursive Types

We can also define "real" lists. These cannot be reflected because we don't want a full compiler in the system.

```
instance (ReifyReflect t)=>
         ReifyReflect [t] where
 rr =
  RR(λi v → foldr Cons Nil (map (λx → reif i x) v),
     error "Cannot␣reflect␣recursive␣type␣(List)!")
  where
   RR(reif,refl) = rr :: ReifyReflect t=>RR t
```

2.7 Partial evaluation

Partial evaluation is merely reifying a value since all the static reductions are done by the (compiled) Haskell code!

```
tdpe v = reify 0 v
 where RR(reify,_) = rr :: ReifyReflect t=>RR t
```

2.8 Printing

Printing expressions uses the Haskell precedence rules to get the parentheses right.

```
instance Show Expr where
 showsPrec n e =
  case e of
   Var x       → shows x . ss"␣"
   Lambda x e  → spp 0 (ss"\\"␣.␣shows␣x␣.␣ss" → "␣.␣sp␣1␣e)
   Apply e1 e2 → spp 2 (sp 2 e1 . ss"␣" . sp 3 e2)
   Base s      → ss s
   Pair e1 e2  → spp 0 (ss"(" . sp 0 e1 . ss"," . sp 0 e2 . ss")")
   Fst e       → spp 2 (ss"fst␣" . sp 3 e)
   Snd e       → spp 2 (ss"snd␣" . sp 3 e)
   Nil         → ss"[]"
   Cons e1 e2  → spp 1 (sp 2 e1 . ss":" . sp 1 e2)
  where
   spp n' s | n ≤ n'     = s
            | otherwise = ss"(" . s . ss")"
   sp = showsPrec
   ss = showString

instance Show Vr where
 showsPrec _ (Vr v) = showString (v++"␣")
```

1. Exercise (user-declared product type). Say that a user declares a new (non-recursive) data type with

$$\texttt{newtype}\ t\ \texttt{=}\ c\ t_1\ \ldots\ t_n$$

what should be added in the user's code to permit reifying of this new data type?

2. Exercise (inductive trees). Represent *finite trees* in a way similar to Church lists and show that you can produce code mapping a function over all leaves of such a tree.

3. Exercise (user-declared sum types). *Research level.* Think about how one can make reification of the simplest sum type, namely `Bool`, work, based on the definitions:

```
data Bool = False | True
```

4. Exercise (user-declared inductive types). *Research level.* Think about how one can code reification of an inductive variants of user-defined recursive types. (Hint: Try to derive the Church inducer by automatic means.)

References

[1] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt and David E. Rydeheard, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199. Springer-Verlag, 1995.

[2] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for a polymorphic system. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[3] Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993.

[4] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[5] Djordje *C*ubrić, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 1997. To appear.

[6] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[7] Olivier Danvy and Kristoffer Høgsbro Rose. Higher-order rewriting and partial evaluation. In Tobias Nipkow, editor, *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Kyoto, Japan, March 1998. Springer-Verlag. Extended version available as the technical report BRICS-RS-97-46.

[8] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.