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 (with-
out any sum-types, i.e., Bool or types made with |).

1.1 Definition (2-level functional programming). The “2-level” \( \lambda \)-terms
are given by the inductive definition (or abstract syntax)

\[
T ::= B \mid T_1 \rightarrow T_2 \mid T_1 \times T_2 \\
v ::= C \mid x \mid \lambda x. v \mid v_0 \rightarrow v_1 \mid \text{pair}(v_1, v_2) \mid \text{fst}(v) \mid \text{snd}(v) \\
e ::= C \mid x \mid \lambda x. e \mid e_0 \mid e_1 \mid \text{pair}(e_1, e_2) \mid \text{fst}(e) \mid \text{snd}(e)
\]

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:

\[
(\lambda x. v[x]), X \rightarrow v[X] \hspace{1cm} (\beta)
\]

\[
\text{fst}(\text{pair}(v_1, v_2)) \rightarrow v_1 \hspace{1cm} (1)
\]

\[
\text{snd}(\text{pair}(v_1, v_2)) \rightarrow v_2 \hspace{1cm} (2)
\]

---

*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
1.2. Definition (2-level $\eta$-expansion).

\[
\downarrow^B(v) \rightarrow v \quad \text{($\downarrow^B$)}
\]

\[
\downarrow^{T_1 \rightarrow T_2}(v) \rightarrow \lambda x. \downarrow^{T_2}(\overline{\uparrow^{T_1}(x)}) \quad \text{($\downarrow^\rightarrow$)}
\]

\[
\downarrow^{T_1 \times T_2}(v) \rightarrow \text{pair}(\downarrow^{T_1}(\text{fst}(v)), \downarrow^{T_2}(\text{snd}(v))) \quad \text{($\downarrow^\times$)}
\]

\[
\uparrow^B(E) \rightarrow E \quad \text{($\uparrow^B$)}
\]

\[
\uparrow_{T_1 \rightarrow T_2}(E) \rightarrow \overline{\lambda v. \uparrow_{T_2}(E(\downarrow^{T_1}(v)))} \quad \text{($\uparrow^\rightarrow$)}
\]

\[
\uparrow_{T_1 \times T_2}(E) \rightarrow \text{pair}(\uparrow^{T_1}(\text{fst}(E)), \uparrow^{T_2}(\text{snd}(E))) \quad \text{($\uparrow^\times$)}
\]

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 $\eta$-expansion, or (standard) type-directed partial evaluation (tdpe) of a simple Haskell subset.

1 module TDPE where

2.1 Expressions

Expressions are data values of the following obvious type.

\[
\text{data Expr} = \text{Var} \text{ Vr} \quad \text{-- lambda terms}
\]

\[
\mid \text{Lambda} \text{ Vr} \text{ Expr}
\]

\[
\mid \text{Apply} \text{ Expr} \text{ Expr}
\]

\[
\mid \text{Base} \text{ String} \quad \text{-- base values}
\]

\[
\mid \text{Pair} \text{ Expr} \text{ Expr} \quad \text{-- product type}
\]

\[
\mid \text{Fst} \text{ Expr}
\]

\[
\mid \text{Snd} \text{ Expr}
\]

\[
\mid \text{Nil} \quad \text{-- inductive list type}
\]

\[
\mid \text{Cons} \text{ Expr} \text{ Expr}
\]

(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 ("\lambda" cannot be added as we cannot extend the syntax of Haskell):

11 apply x y = x y

12 pair x y = (x,y)

13 nil = []

14 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 $\eta$-expansion is defined by two mutually recursive functions, one reifying values to expressions and the other reflecting expressions to values, corresponding to $\downarrow(\cdot)$ and $\uparrow(\cdot)$ of the introduction, respectively. Both take a first argument 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”):

```haskell
    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 $\text{rr}$ for every type.

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

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

```haskell
    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 = lambda v -> 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 $\text{Alpha}$, $\text{Beta}$, ..., $\text{Omega}$. These are just aliased to the $\text{Expr}$ type to make the reification be the identity on types as dictated by the definition.

```haskell
    instance ReifyReflect Expr where
        rr = RR(\v -> v, \e -> e)
```
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.

```haskell
e6 instance ReifyReflect () where
e7     rr =
e8     RR(λi v → Base "()",
  e9     error "Cannot reflect base value u::().")
```

Integers are also merely printed.

```haskell
e6 instance ReifyReflect Integer where
e7     rr =
e8     RR(λi v → Base (show v),
  e9     error "Cannot reflect base value u::Integer.")
```

2.4 Product Types

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

```haskell
e6 instance (ReifyReflect alpha, ReifyReflect beta) =>
e7     ReifyReflect (alpha, beta) where
e8     rr = RR(reif, refl) where
```
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).

```haskell
71 reif i v = Pair (reif1 i (fst v)) (reif2 i (snd v))
72 refl i e = pair (refl1 i (fst e)) (refl2 i (snd e))
73 RR(reif1,refl1) = rr :: ReifyReflect alpha => RR alpha
74 RR(reif2,refl2) = rr :: ReifyReflect beta => RR 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.

```haskell
94 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!

```haskell
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.

```haskell
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"("_ _ 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

   newtype t = c t1 ... 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:

   ```haskell
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 inductor by automatic means.)

References


