

Accumulating bindings

Sam Lindley
The University of Edinburgh
Sam.Lindley@ed.ac.uk

Abstract

We give a Haskell implementation of Filinski’s normalisation by evaluation algorithm for the computational lambda-calculus with sums. Taking advantage of extensions to the GHC compiler, our implementation represents object language types as Haskell types and ensures that type errors are detected statically.

Following Filinski, the implementation is parameterised over a residualising monad. The standard residualising monad for sums is a continuation monad. Defunctionalising the uses of the continuation monad we present the binding tree monad as an alternative.

1 Introduction

Filinski [12] introduced normalisation by evaluation for the computational lambda calculus, using layered monads [11] for formalising name generation and for collecting bindings. He extended his algorithm to handle products and sums, and outlined how to prove correctness using a Kripke logical relation. Filinski’s algorithm is parameterised by a *residualising* monad that is used for interpreting computations.

In the absence of sums he gives two concrete residualising monads: one a continuation monad and the other an accumulation monad over a list of bindings, henceforth the *binding list monad*. He further shows how by using the internal monad of the metalanguage it is possible to give corresponding algorithms for type-directed partial evaluation. If the metalanguage supports delimited continuations then we can use shift and reset [9] in place of the continuation monad. If the metalanguage supports state then we can use a mutable list of bindings in place of the binding list monad.

Filinski demonstrated how to extend his algorithm to support sums, but only in the case of the continuation monad (or delimited continuations). He writes:

Products could be added to an accumulation-based interpretation without too much trouble,

but sums apparently require the full power of applying a single continuation multiple times.

In my PhD thesis I observed [14, Chapter 4] that by generalising the accumulation-based interpretation from a list to a tree that it is possible to use an accumulation-based interpretation for normalising sums. There I focused on an implementation using the state supported by the internal monad of the metalanguage. The implementation uses Huet’s zipper [13] to navigate a mutable binding tree. Here we present a Haskell implementation of normalisation by evaluation for the computational lambda calculus using a generalisation of Filinski’s binding list monad to incorporate a tree rather than a list of bindings.

(One motivation for using state instead of continuations is performance. We might expect a state-based implementation to be faster than an alternative delimited continuation-based implementation. For instance, Sumii and Kobayashi [17] claim a 3-4 times speed-up for state-based versus continuation-based let-insertion. The results of my experiments [14] suggest that in the case of sums it depends on the low-level implementation of the host language. For SML/NJ, which uses a CPS-based intermediate representation, the delimited continuations-based implementation outperformed the state-based implementation. The inefficiency of the state-based implementation is in part due to it having to duplicate computation (something which seems hard to avoid if we have sums). However, for MLton, which does not use a CPS-based intermediate representation, the state-based implementation was faster. The implementation of delimited continuations in MLton is an order of magnitude slower than that of SML/NJ.)

The main contributions of this article are rather modest:

- A clean implementation of Filinski’s algorithm for normalisation by evaluation for the computational lambda calculus with sums in Haskell.
- A generalisation of the binding list monad from binding lists to bindings trees allowing it to be plugged into Filinski’s algorithm.

2 Implementation

Danvy et al [10] give an implementation of normalisation by evaluation for call-by-name simply-typed lambda-calculus in Haskell. Input terms are represented as closed Haskell expressions. The type-indexed `reify` and `reflect` functions are written as instances of a type class. The output terms are represented as typed higher-order abstract syntax terms in normal form. As the types and structure of normal forms are enforced by the Haskell type system they can be sure that their algorithm: a) preserves typing and b) outputs normal forms.

In this section we use some similar ideas, but our goals are slightly different. Our implementation is for call-by-value computational lambda-calculus with sums. Our implementation takes typed HOAS as input and outputs FOAS terms in normal form. We use a GADT in conjunction with a type class in order to explicitly represent Haskell types as terms. Thus we can leverage the Haskell type checker to statically check that the input term is well-typed and that its type matches up with the input type explicitly supplied to the normalisation function. As in the Danvy et al’s implementation, it is necessary to explicitly write the type of the term being normalised somewhere, as the type Haskell will infer will be polymorphic, whereas our object language is monomorphic.

The full Haskell source is available at the following URL:
<http://homepages.inf.ed.ac.uk/slindley/nbe/nbe-sums.hs>.

We begin by defining an algebraic datatype for representing computational lambda calculus terms extended with sums.

```
type Var = String
```

```
data Exp = Var Var
         | Lam Var Exp
         | App Exp Exp
         | Inl Exp
         | Inr Exp
         | Case Exp Var Exp Var Exp
         | Let Var Exp Exp
```

We use the Haskell `String` type to represent variables. Terms are constructed from variables, lambda, application, left injection, right injection, case and let. As Filinski remarks, let is redundant; it is included in order to give nicer normal forms. We chose not to include products in the presentation because there is little of interest to say about them and it is straightforward to add them.

The first-order `Exp` datatype is rather a verbose way of writing syntax and makes no guarantees about binding. We choose to use higher-order abstract syntax as our input syntax, which automatically restricts us to closed terms and makes use of Haskell’s built-in binding. Following

Carette et al [8] we use a type class, relying on parametricity to exclude so-called “exotic terms” [5].

```
class CompLam exp where
  lam :: (exp → exp) → exp
  app :: exp → exp → exp
  inl :: exp → exp
  inr :: exp → exp
  case_ ::
    exp → (exp → exp) → (exp → exp) → exp
  let_ :: exp → (exp → exp) → exp
```

```
type Hoas = ∀exp . CompLam exp ⇒ exp
```

Following Atkey et al [6] it is straightforward to convert from the HOAS representation to the FOAS representation by defining a suitable instance of the `CompLam` type class.

```
hoasToExp :: Hoas → Exp
hoasToExp v = evalGen v 0
```

```
instance CompLam (Gen Exp) where
  lam f = do x ← nextName
           e ← f (return (Var x))
           return$ Lam x e
  v1 ‘app’ v2 = do e1 ← v1
                  e2 ← v2
                  return$ App e1 e2
  inl v = do e ← v
            return$ Inl e
  inr v = do e ← v
            return$ Inr e
  case_ v l r = do e ← v
                  x1 ← nextName
                  x2 ← nextName
                  e1 ← l (return (Var x1))
                  e2 ← r (return (Var x2))
                  return$ Case e x1 e1 x2 e2
  let_ v f = do e ← v
               x ← nextName
               e’ ← f (return (Var x))
               return$ Let x e e’
```

Instead of outputting a de Bruijn representation, here we choose to use a name generation monad, targeting a representation with explicit names in order to fit in with Filinski’s monadic treatment of names.

```
type Gen = State Int
```

```
nextName :: Gen Var
nextName =
  do { i ← get; put (i+1); return ("x" ++ show i) }
```

```
evalGen :: Gen a → Int → a
evalGen = evalState
```

The simple types of our source language are given by the grammar

$$\sigma, \tau ::= A \mid B \mid C \mid \sigma \rightarrow \tau \mid \sigma + \tau$$

where A, B, C are abstract base types. In Haskell, for $+$ we write \oplus (in order to avoid clashing with the built-in $+$) which is syntactic sugar for the `Either` datatype.

```
data A
data B
data C
```

```
type a  $\oplus$  b = Either a b
```

Following Atkey et al [6], we define a GADT `Rep` of representations of simple types along with a typeclass `Representable a` which allows us to smoothly bridge the gap between Haskell types and object language type representations.

```
data Rep :: * -> * where
  A :: Rep A
  B :: Rep B
  C :: Rep C
  (~>) :: Rep a -> Rep b -> Rep (a -> b)
  ( $\oplus$ ) :: Rep a -> Rep b -> Rep (a  $\oplus$  b)
```

```
class Representable a where
  rep :: Rep a
```

```
instance Representable A where rep = A
instance Representable B where rep = B
instance Representable C where rep = C
```

```
instance (Representable a, Representable b) =>
  Representable (a -> b) where
  rep = rep ~> rep
```

```
instance (Representable a, Representable b) =>
  Representable (a  $\oplus$  b) where
  rep = rep  $\oplus$  rep
```

For instance, we can now write down a Haskell term representing the type

$$((A \rightarrow B) \rightarrow A) \rightarrow A$$

as

```
rep :: Rep ((A -> B) -> A) -> A
```

Note that the `Representable` type class is closed by construction, as the `Rep` GADT only admits simple type representations.

2.1 Residualising monads

Filinski's algorithm is parameterised by a monad, called a *residualising* monad. The idea is that a residualising monad will be used to interpret computations and that it must contain enough hooks in order to allow us to recover syntax from the semantics. Filinski gives a fairly abstract characterisation of residualising monads in terms of several operations. We capture his notion via a typeclass of residualising monads.

```
class Monad m => Residualising m where
  gamma :: Gen a -> m a
  collect :: m Exp -> Gen Exp
  bind :: Exp -> m Var
  binds :: Exp -> m (Var  $\oplus$  Var)
```

Filinski assumes that a residualising monad is layered atop [11] a name generation monad.

The `gamma` operation is a monad morphism lifting a computation of type a in the name generation monad to a computation of type a in the residualising monad.

The `bind` and `binds` operations respectively introduce `let` and `case` bindings, by storing the bound term and returning its name inside the residualising monad.

The `collect` operation collects all the bindings from the residualising monad.

The `collect`, `bind` and `binds` operations must satisfy the following equations.

```
collect (return e)      = return e
collect (bind e>>=f) =
  do x <- nextName
     e' <- collect (f x)
     return (Let x e e')
collect (binds e>>=f) =
  do x1 <- nextName
     x2 <- nextName
     e1 <- collect (f (Left x1))
     e2 <- collect (f (Right x2))
     return (Case e x1 e1 x2 e2)
```

These equations give a rather direct correspondence between the syntactic and semantic representations of bindings. One can construct a tree of bindings in the semantics using `bind`, `binds`, `>>=` and `return`, and then reify it as syntax. For instance:

```
collect (do s <- binds e
         case s of
           Left x1 ->
             do z <- bind e'; return (Var z)
           Right x2 -> return (Var x2))
=
do x1 <- nextName
   x2 <- nextName
   z <- nextName
   return (Case e x1 (Let z e' (Var z)) x2 (Var x2))
```

2.2 A monadic evaluator

The evaluator is a standard evaluator for the computational lambda-calculus parameterised by a monad as well as operations for boxing and unboxing functions and sums. In order to perform normalisation by evaluation we instantiate it with a residualising monad.

```
type Env a = [(Var, a)]
```

```
empty :: Env a
empty = []

extend :: [(Var, a)] → Var → a → [(Var, a)]
extend env x v = (x, v):env
```

```
class FunInt v m where
  injFun :: (v → m v) → m v
  projFun :: v → (v → m v)
```

```
class SumInt v where
  injSum :: v ⊕ v → v
  projSum :: v → v ⊕ v
```

```
eval ::
  (Monad m, FunInt a m, SumInt a) ⇒
  Env a → Exp → m a
eval env (Var x) =
  return (fromJust (lookup x env))
eval env (Lam x e) =
  injFun (λv → eval (extend env x v) e)
eval env (App e1 e2) =
  do
    v1 ← eval env e1
    v2 ← eval env e2
    projFun v1 v2
eval env (Let x e1 e2) =
  do
    v ← eval env e1
    eval (extend env x v) e2
eval env (Inl e) =
  do
    v ← eval env e
    return (injSum (Left v))
eval env (Inr e) =
  do
    v ← eval env e
    return (injSum (Right v))
eval env (Case e x1 e1 x2 e2) =
  do
    v ← eval env e
    case projSum v of
      Left v → eval (extend env x1 v) e1
      Right v → eval (extend env x2 v) e2
```

2.3 The normalisation function

We define the semantics in Haskell using a datatype `SemV m` for values and a datatype `SemC m` for computations. The parameter `m` is the residualising monad.

```
data SemV m = Neutral Exp
  | Fun (SemV m → SemC m)
  | Sum (SemV m ⊕ SemV m)
type SemC m = m (SemV m)
```

Base types are interpreted as expressions, functions as Haskell functions from values to computations, and sums

using the Haskell `Either` datatype. Computations are interpreted in the residualising monad. The boxing and unboxing functions are straightforward.

```
instance Residualising m ⇒ FunInt (SemV m) m where
  injFun f = return (Fun f)
  projFun = λ(Fun f) → f
```

```
instance Residualising m ⇒ SumInt (SemV m) where
  injSum = Sum
  projSum = λ(Sum s) → s
```

Each residualising monad `m` gives rise to a residualising evaluator.

```
type ResEval m = Env (SemV m) → Exp → SemC m
```

Having pinned down the interpretation, we now need to define a function `reifyC` mapping semantic computations to syntactic normal forms. The `reifyC` function is type-indexed. The supplied computation is run inside the residualising monad, binding the result to a value which is reified as an expression with the function `reifyV` lifted into the residualising monad by `gamma`. Any bindings that are generated are collected through the `collect` function.

```
reifyC ::
  Residualising m ⇒ Rep a → SemC m → Gen Exp
reifyC a c = collect (do v ← c; gamma (reifyV a v))
```

The `reifyV` function (in conjunction with its partner, the `reflectV` function) does most of the actual work. It follows the usual pattern for normalisation by evaluation. On base types it is the identity (modulo unboxing and lifting into the name generation monad). On functions it generates a fresh name for a lambda expression that is reflected as a value and fed into the function before reifying the result. On sums it does a case split on the supplied value, reifying at the appropriate type according to whether it is a left or a right injection.

```
reifyV ::
  Residualising m ⇒ Rep a → SemV m → Gen Exp
reifyV A (Neutral e) = return e
reifyV B (Neutral e) = return e
reifyV C (Neutral e) = return e
reifyV (a ~> b) (Fun f) =
  do x ← nextName
    e ← reifyC b (do v ← reflectV a x; f v)
    return$ Lam x e
reifyV (a ⊕ b) (Sum (Left v)) =
  do e ← reifyV a v
    return$ Inl e
reifyV (a ⊕ b) (Sum (Right v)) =
  do e ← reifyV b v
    return$ Inr e
```

As the body of a function in the semantics is a computation, we call `reifyC` here instead of `reifyV`.

Of course, we also need to define the function `reflectV` for reflecting neutral expressions as semantics. In fact, the only neutral value expressions we have are variables, so we specialise the type of `reflectV` to take a variable rather than an expression. Again `reflectV` follows the usual pattern for normalisation by evaluation. At base types it gives the variable itself. At function types it gives a function that returns the result of reflecting the input variable applied to the reified argument of the function. At sum type it calls `binds` on the input variable and then performs a case split reflecting at the appropriate type according to whether the value bound by `binds` is a left or a right injection.

```
reflectV ::
  Residualising m => Rep a -> Var -> SemC m
reflectV A x = return (Neutral (Var x))
reflectV B x = return (Neutral (Var x))
reflectV C x = return (Neutral (Var x))
reflectV (a ~> b) x =
  return (Fun (\v -> do e <- gamma (reifyV a v)
              reflectC b x e))
reflectV (a @ b) x =
  do v <- binds (Var x)
  case v of
    Left x1 ->
      do v1 <- reflectV a x1
      return (Sum (Left v1))
    Right x2 ->
      do v2 <- reflectV b x2
      return (Sum (Right v2))
```

For the body of a function we need to reflect a computation expression in the form of a variable applied to an expression. The `reflectC` function binds the application before calling `reflectV` on the resulting variable.

```
reflectC ::
  Residualising m => Rep a -> Var -> Exp -> SemC m
reflectC a x e =
  do x <- bind (App (Var x) e)
  reflectV a x
```

We are now in a position to define a normalisation function.

```
normU ::
  Residualising m =>
  ResEval m -> Rep a -> Hoas -> Exp
normU eval a e =
  evalGen (reifyC a (eval empty (hoasToExp e))) 0
```

The function `normU` takes a residualising evaluator, a type and a HOAS term, and returns a FOAS term in normal form. The first argument is a hack to allow us to choose different residualising monads at run-time. We will always pass in the same `eval` function, but with a different type annotation in order to tell GHC which residualising monad to use.

2.4 Two residualising monads

Filinski gives two residualising monads for the computational lambda-calculus without sums: a continuation monad with answer type `Gen Exp` and an accumulation monad over lists of let bindings, with name generation layered atop it.

He shows how to use the former to handle sums. Here, we also show how to generalise the latter to handle sums.

The continuation monad The continuation monad is built into Haskell. We just need to instantiate it at the appropriate answer type, and then define the residualising operations.

```
type ContGenExp = Cont (Gen Exp)

instance Residualising ContGenExp where
  gamma f = Cont (\k -> do m <- f; k m)
  collect (Cont f) = f return
  bind e =
    Cont (\k ->
      do x <- nextName
      e' <- k x
      return (Let x e e'))
  binds e =
    Cont (\k ->
      do x1 <- nextName
      x2 <- nextName
      e1 <- k (Left x1)
      e2 <- k (Right x2)
      return (Case e x1 e1 x2 e2))
```

The binding tree monad The datatype underlying Filinski's binding list monad can be expressed in Haskell as follows.

```
data Acc' a = Val a
            | LetB Var Exp (Acc' a)
```

This encodes a binding list (of let bindings) alongside a value of type `a`.

In order to extend `ACC'` to handle sums we need to accumulate trees rather than lists of bindings. The tree structure arises from case bindings which bind one variable for each of the two branches of a case. Rather than accumulating a binding list alongside a single value, we now accumulate a binding tree alongside a list of values — one for each leaf of the tree.

```
data Acc a = Val a
            | LetB Var Exp (Acc a)
            | CaseB Exp Var (Acc a) Var (Acc a)
```

The nodes of the tree are let and case bindings and the leaves are values. It is now straightforward to define a monad instance for `ACC`.

```

instance Monad Acc where
  return v = Val v
  Val v >>= f = f v
  LetB x e m >>= f =
    LetB x e (m >>= f)
  CaseB e x1 m1 x2 m2 >>= f =
    CaseB e x1 (m1 >>= f) x2 (m2 >>= f)

```

The `>>=` operator is recursively defined over the continuations of each binding. The operation `t >>= f` simply descends to the leaves replacing each leaf `Val v` with the tree `f v`.

It is worth noting that both the binding list monad and the binding tree monad are *free monads* [18]. The former is the free monad over the functor underlying the datatype:

```

data L a = LetL Var Exp a

```

and the latter is the free monad over the functor underlying the datatype:

```

data T a = LetT Var Exp a
         | CaseT Exp Var a Var a

```

The connection with free monads is unsurprising given that NBE hinges on including enough syntactic hooks in a denotational semantics, and free monads constitute prototypical “syntactic” monads.

If the values at the leaves of a tree are expressions, then we can flatten the tree to a single expression.

```

flatten :: Acc Exp → Exp
flatten (Val e) = e
flatten (LetB x e t) = Let x e (flatten t)
flatten (CaseB v x1 t1 x2 t2) =
  Case v x1 (flatten t1) x2 (flatten t2)

```

In order to obtain a residualising monad we layer the name generation monad atop the binding tree monad.

```

newtype GenAcc a = GA {unGA :: Gen (Acc a)}
instance Monad GenAcc where
  return = GA . return . return
  m >>= k =
    GA (do c ← unGA m
        case c of
          Val v → unGA (k v)
          LetB x e m →
            do t ← unGA (GA (return m) >>= k)
               return (LetB x e t)
          CaseB e x1 m1 x2 m2 →
            do t1 ← unGA (GA (return m1) >>= k)
               t2 ← unGA (GA (return m2) >>= k)
               return (CaseB e x1 t1 x2 t2))

```

Now we can define the residualising operations.

```

instance Residualising GenAcc where
  gamma f = GA (do v ← f; return (return v))
  collect (GA f) = do t ← f
                    return (flatten t)

```

```

bind e =
  GA (do x ← nextName
        return$ LetB x e (Val x))
binds e =
  GA (do x1 ← nextName
        x2 ← nextName
        return$ CaseB e x1 (Val (Left x1))
                x2 (Val (Left x2)))

```

Notice the similarity between these definitions and those for the continuation monad. In essence they do the same thing. Where the continuation monad manipulates bindings implicitly using functional continuations, the binding tree monad manipulates them explicitly using a binding tree data structure. The binding tree is really just a defunctionalised [16] version of the continuation monad.

Having defined two residualising monads we are now in a position to instantiate our normalisation function with either of them:

```

normAccU = normU (eval :: ResEval GenAcc)
normContU = normU (eval :: ResEval ContGenExp)

```

Examples

```

> normAccU (rep :: Rep (A → A)) (lam (λx → x))
λx0 → x0
> normContU (rep :: Rep (A → A)) (lam (λx → x))
λx0 → x0

```

```

> normAccU
  (rep :: Rep (A ⊕ B → A ⊕ B))
  (lam (λx → x))
λx0 → case x0 of
  Left x1 → Left x1;
  Right x2 → Right x2

> normContU
  (rep :: Rep ((A ⊕ (B → C)) → (A ⊕ (B → C))))
  (lam (λx → x))
λx0 → case x0 of
  Left x1 → Left x1;
  Right x2 →
    Right (λx3 → let x4 = x2 x3 in x4)

```

Unfortunately type errors are manifested as runtime errors:

```

> normAccU
  (rep :: Rep (A → B → A))
  (lam (λx → x))
λx0 → *** Exception: tacc.hs:(306,0)-(318,19):
      Non-exhaustive patterns in function reifyV

```

```

> normAccU
  (rep :: Rep (A → A))
  (lam (λx → app x x))
λx0 → *** Exception: tacc.hs:361:14-26:
      Non-exhaustive patterns in lambda

```

Fortunately, it is easy to do better.

2.5 Static typing

Though we have successfully demonstrated how to use a Haskell type as the argument to the normalisation function our current implementation does not check either that the input term is well-typed or that the supplied type matches up with the type of the input term, whose representation is untyped.

Following Atkey et al [6], we augment our HOAS representation with type information, taking advantage of parametricity and our encoding of representable types to preclude “exotic types”.

```
class TCompLam exp where
  tlam :: (Representable a, Representable b) =>
    (exp a -> exp b) -> exp (a -> b)
  tapp :: (Representable a, Representable b) =>
    exp (a -> b) -> exp a -> exp b
  tinl :: (Representable a, Representable b) =>
    exp a -> exp (a ⊕ b)
  tinr :: (Representable a, Representable b) =>
    exp b -> exp (a ⊕ b)
  tcase :: (Representable a, Representable b,
    Representable c) =>
    exp (a ⊕ b) -> (exp a -> exp c) ->
    (exp b -> exp c) -> exp c
  tlet :: (Representable a, Representable b) =>
    exp a -> (exp a -> exp b) -> exp b
```

```
type THoas a =
  ∀(exp :: * -> *) . TCompLam exp => exp a
```

For now, our goal is to ensure that the type of the representation passed to the normalisation function matches up with the type of the HOAS input term. We are not attempting to use the Haskell type system to enforce well-typedness of the resulting FOAS representation.

(It may be possible to push the types further. Atkey [3] has made some progress in adapting the code from this article to take typed HOAS terms to typed HOAS terms in normal form, along the lines of Danvy et al [10]. His solution relies on making a syntactic distinction between value expressions and computation expressions and a somewhat more complicated datatype for higher-order abstract syntax.)

Thus we instantiate TCompLam with a newtype that simply forgets its type argument.

```
thoasToExp :: THoas a -> Exp
thoasToExp v = evalGen (unT v) 0
```

```
newtype T a = T {unT :: Gen Exp}
```

```
instance TCompLam T where
  tlam f = T$
```

```
    do x ← nextName
       e ← unT$ f (T$ return (Var x))
       return (Lam x e)
v1 'tapp' v2 = T$
  do e1 ← unT v1
     e2 ← unT v2
     return (App e1 e2)
tinl v = T$
  do e ← unT v
     return (Inl e)
tinr v = T$
  do e ← unT v
     return (Inr e)
tcase v l r = T$
  do e ← unT v
     x1 ← nextName
     x2 ← nextName
     e1 ← unT$ l (T$ return (Var x1))
     e2 ← unT$ r (T$ return (Var x2))
     return (Case e x1 e1 x2 e2)
tlet v f = T$
  do e ← unT v
     x ← nextName
     e' ← unT$ f (T$ return (Var x))
     return (Let x e e')
```

It would be nice if we could get rid of the boxing (T) and unboxing (unT) in the above instance. If Haskell had better support for parameterised monads [4] then this would be straightforward.

Now we can define normalisation functions that statically detect type errors both in the input term and between the type of the input term and the type representation argument.

```
norm ::
  Residualising m =>
  ResEval m -> Rep a -> THoas a -> Exp
norm eval a e =
  evalGen (reifyC a (eval empty (thoasToExp e))) 0
```

```
normAcc = norm (eval :: ResEval GenAcc)
normCont = norm (eval :: ResEval ContGenExp)
```

The type errors from the previous examples are now reported as type errors.

```
> normAcc
(rep :: Rep (A -> B -> A))
(tlam (λx -> x))
```

```
<interactive>:1:48:
Couldn't match expected type 'B -> A'
against inferred type 'A'
  Expected type: exp (B -> A)
  Inferred type: exp A
In the expression: x
In the first argument of 'tlam',
namely '(λ x -> x)'
```

```

> normAcc
  (rep :: Rep (A → A))
  (tlam (λx → tapp x x))

<interactive>:1:48:
  Couldn't match expected type 'a → A'
  against inferred type 'A'
    Expected type: exp (a → A)
    Inferred type: exp A
  In the first argument of 'tapp', namely 'x'
  In the expression: tapp x x

```

3 Closing remarks

Internalising the binding tree monad It is possible to internalise the accumulation-based implementation of normalisation by evaluation for the computation lambda calculus with sums. The idea is to simulate the binding tree using the state monad. Internalising the binding list monad is easy, and well-established as a means for implementing type-directed partial evaluation in ML. One simply stores the list of bindings in a state cell instead of the binding list monad.

Things get more complicated with the binding tree monad because although we can still store the binding tree in a state cell, we now also have to handle multiple values in tandem with the binding tree. To account for the multiple values, the `collect` function must run its argument multiple times for each branch of the binding tree. As well as storing the binding tree we also need to store which branch of the binding tree we are currently in, i.e., which value we are currently computing.

The binding tree, along with the current path, can be represented using a zipper structure. A single state cell is used to store the cursor into the binding tree (see [14, Chapter 4] and the code accompanying this article for further details).

Sums for call-by-name Sums are considerably harder to handle in the pure call-by-name setting (vanilla simply-typed lambda calculus). One reason why we can handle them reasonably smoothly in the computational lambda-calculus is that we interpret everything in a residualising monad anyway, so we can “squirrel away” the bindings in the monad. Another reason is that in computational lambda calculus normal forms every application subterm is explicitly bound. If such a term has sum type then we can always eliminate it immediately after it is bound using a case split.

In contrast, in the call-by-name setting there is no monad in the standard interpretation of terms. Furthermore, application subterms are not explicitly named and may in fact be pure, so it is less clear where to insert a case split and one has to be careful about managing redundant case splits.

Nevertheless, Altenkirch et al [2] have described a normalisation by evaluation algorithm for call-by-name lambda calculus using categorical techniques, and Balat et al [7] have implemented type-directed partial evaluation for sums using powerful delimited continuations operators. Lindley [15] has given rewrite rules and shown that simply-typed lambda calculus with sums is confluent and strongly normalising. Altenkirch and Chapman [1] advocate a “big-step” approach lying somewhere between small-step rewriting and full-on normalisation by evaluation. It would be interesting to connect these separate lines of work.

References

- [1] T. Altenkirch and J. Chapman. Big-step normalisation. *JFP*, 19(3 & 4):311–333, 2009.
- [2] T. Altenkirch, P. Dybjer, M. Hofmann, and P. J. Scott. Normalization by evaluation for typed lambda calculus with co-products. In *LICS*, pages 303–310, 2001.
- [3] R. Atkey, 2009. Personal communication.
- [4] R. Atkey. Parameterised notions of computation. *JFP*, 19(3 & 4):335–376, 2009.
- [5] R. Atkey. Syntax for free: Representing syntax with binding using parametricity. In *TLCA*, 2009. To appear.
- [6] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain specific languages. In *Haskell*, 2009. To appear.
- [7] V. Balat, R. D. Cosmo, and M. P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *POPL*, pages 64–76, 2004.
- [8] J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated. *JFP*, 2009. To appear.
- [9] O. Danvy and A. Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.
- [10] O. Danvy, M. Rhiger, and K. H. Rose. Normalization by evaluation with typed abstract syntax. *J. Funct. Program.*, 11(6):673–680, 2001.
- [11] A. Filinski. Representing layered monads. In *POPL*, pages 175–188, 1999.
- [12] A. Filinski. Normalization by evaluation for the computational lambda-calculus. In *TLCA*, pages 151–165, 2001.
- [13] G. P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.
- [14] S. Lindley. *Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages*. PhD thesis, University of Edinburgh, 2005.
- [15] S. Lindley. Extensional rewriting with sums. In *TLCA*, pages 255–271, 2007.
- [16] J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.
- [17] E. Sumii and N. Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2-3):101–142, 2001.
- [18] W. Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.