Evaluation à la carte
non-strict evaluation via compositional data types
Bahr, Patrick

Published in:
Proceedings of the 23rd Nordic Workshop on Programming Theory

Publication date:
2011

Document version
Peer reviewed version

Citation for published version (APA):
Evaluation à la Carte
Non-Strict Evaluation via Compositional Data Types

Patrick Bahr
Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen, Denmark
paba@diku.dk

Abstract
We describe how to perform monadic computations over recursive data structures with fine grained control over the evaluation strategy. This solves the issue that the definition of a recursive monadic function already determines the evaluation strategy due to the necessary sequencing of the monadic operations. We show that compositional data types already provide the structure needed in order to delay monadic computations at any point of the computation.

1 Introduction
Algebraic data types offer an excellent representation of abstract syntax trees (ASTs). The ease with which functional programming languages allow us to manipulate algebraic data types makes the functional programming paradigm a powerful tool for performing transformations on ASTs – an ubiquitous task when writing compilers and interpreters.

A simple example is the following Haskell definition of an algebraic data type representing a simple expression language over integers and pairs:

```
data Exp = Const Int | Pair Exp Exp |
        | Add Exp Exp | Fst Exp | Snd Exp
```

Apart from the constructors for integers and pairs, the language contains addition and the projection operators \(Fst\) and \(Snd\). Implementing an evaluation function for this language is a simple exercise:

```
eval :: Exp \to Exp
eval (Const i) = Const i
eval (Pair x y) = \text{Pair} (\text{eval} x) (\text{eval} y)
eval (Add x y) = \text{case} (\text{eval} x, \text{eval} y) \of
    (\text{Const} i, \text{Const} j) \mapsto \text{Const} (i + j)
eval (Fst p) = \text{case} \text{eval} p \text{ of} \text{Pair} x y \mapsto x
\text{eval} (\text{Snd} p) = \text{case} \text{eval} p \text{ of} \text{Pair} x y \mapsto y
```

While this function performs the desired evaluation, its type is not as precise as we would expect. According to its type, \(\text{eval}\) produces an expression of type \(\text{Exp}\) which potentially can contain additions and projections. This can be solved by using as codomain of \(\text{eval}\) a separate type \(\text{Value}\) that only contains (copies of) the constructors \(\text{Const}\) and \(\text{Pair}\). This means, however, that also code that works on both \(\text{Exp}\) and \(\text{Value}\) has to be duplicated, e.g. pretty printing and parsing.

2 Data Types à la Carte
Swierstra’s data types à la carte [5] offer an elegant solution to this problem by representing expression types as a fixed point of a functor:

```
data Term f = Term f (Term f)
```

This approach makes it possible to define the signature of the expression language in two components – values and operations – and combine them via the formal sum \(\oplus\) of functors:

```
data (f \oplus g) e = \text{Inl} (f e) | \text{Inr} (g e)
```

We can then define the signatures of our expression language as follows:

```
data Value e = Const Int | Pair e e
data Op e = Add e e | Fst e | Snd e
type Sig = Op \oplus Value
```

This allows us to represent values and expressions as \(\text{Term Value}\) and \(\text{Term Sig}\), respectively.

In addition, Swierstra also defines a binary type class \(\prec\) on signature functors that approximates inclusion. That is, \(f \prec g\) if \(g\) is equal to \(f\) or contains
it as a summand. Most importantly, this type class provide a function to inject a “smaller” signature into a “bigger” one:

\[ \text{inject} :: (f < g) \Rightarrow f \left( \text{Term } g \right) \rightarrow \text{Term } g \]

Functions of the form \( \text{Term } f \rightarrow r \) are written as catamorphisms induced by algebras, i.e. functions of type \( f \rightarrow f \rightarrow r \). This allows us to write functions on a per signature basis, which is achieved by using a type class:

```haskell
class Eval f where
evalAlg :: f (Term Value) \rightarrow Term Value
```

The instantiation of this class for values is trivial:

```haskell
instance Eval Value where
evalAlg = inject
```

For operator symbols we have to provide an implementation that evaluates the arguments appropriately:

```haskell
instance Eval Op where
evalAlg (Add x y) = case (x, y) of
  (Term (Const i), Term (Const j)) \rightarrow inject (Const (i + j))
evalAlg (Fst p) = case p of
  Term (Pair x y) \rightarrow x
evalAlg (Snd p) = case p of
  Term (Pair x y) \rightarrow y
```

Note that the case distinction in the above evaluation algebra as well as in the direct evaluation in Section 1 is incomplete: In case that an argument is not of the expected type, e.g. \( \text{Fst} \) is applied to an integer constant, the evaluation halts with a runtime error.

### 3 Monadic Algebras and Thunks

In order to recover from runtime errors, it is better to use monads to indicate failure explicitly. This can be easily achieved by defining a monadic algebra, i.e. a function of type \( f \rightarrow m \rightarrow r \) for a monad \( m \). Such a monadic algebra gives rise to a monadic catamorphism of type \( \text{Term } f \rightarrow m \rightarrow r \). The evaluation algebra from Section 1 can be easily adapted to such a monadic style. Unfortunately, this will determine the evaluation strategy: The arguments of the operator symbols such as \( \text{Fst} \) have to be evaluated to normal form (in order to determine whether an error occurred). For example, the evaluation of an expression of the form \( \text{fst } (3, \text{snd } 5) \) will yield an error since the evaluation of \( (3, \text{snd } 5) \) to normal form fails due to the second component of the pair.

In order to regain control over the evaluation strategy, we have to allow arbitrary nesting of monadic computations in the result. Instead of the monadic result type \( m \rightarrow r \), we therefore use the result type \( \text{Term } (m \oplus r) \) – making the monad part of the target signature. A monadic computation can thus be embedded into the term structure.

```haskell
thunk :: m (Term (m \oplus f)) \rightarrow Term (m \oplus f)
thunk = inject
```

The evaluation of terms with such thunks to weak head normal form (\( \text{whnf} \)) is implemented by sequencing all thunks until a proper constructor (i.e. in the \( f \)-part of the signature) is reached:

```haskell
whnf :: Monad m \Rightarrow
  Term (m \oplus f) \rightarrow m (f (Term (m \oplus f)))
whnf (Term (Inl m)) = m >>= \text{whnf}
whnf (Term (Inr t)) = return t
```

We can now use this idea to define a non-strict monadic evaluation function using the \( \text{Maybe} \) monad to indicate failure:

```haskell
class EvalT f where
evalAlgT :: f (Term (Maybe \oplus r)) \rightarrow Term (Maybe \oplus r)
```

Again, the case for the value constructors is trivial:

```haskell
instance EvalT Value where
evalAlgT = inject
```

For evaluating the operator symbol applications, we simply evaluate their arguments to \( \text{whnf} \) and create a thunk in the end:

```haskell
evalAlgT (Add x y) = thunk $ do
  Const i \leftarrow \text{whnf } x
  Const j \leftarrow \text{whnf } y
```

2
return (inject (Const (i+j)))
evalAlgT (Fst v) = thunk $ do
    Pair x y ← whnf v
    return x
evalAlgT (Snd v) = thunk $ do
    Pair x y ← whnf v
    return y

By constructing the catamorphism of this algebra, we obtain the following evaluation function

\( \text{evalT} :: \text{Term Sig} \rightarrow \text{Term} (\text{Maybe} \oplus \text{Value}) \)

\( \text{evalT} = \text{cata} \ \text{evalAlgT} \)

With only mild assumptions on the signature functions, we can also easily implement the evaluation to normal form by simply iterating the \( \text{whnf} \) function:

\( \text{nf} :: (\text{Monad } m, \text{Traversable } f) \Rightarrow \text{Term} (m \oplus f) \rightarrow m (\text{Term } f) \)

\( \text{nf} = \text{liftM}
  \text{Term} \ . \text{mapM} \ \text{nf} \ \ll\ll \text{whnf} \)

Eventually, we obtain the desired non-strict evaluation function:

\( \text{eval} :: \text{Term Sig} \rightarrow \text{Maybe} (\text{Term Value}) \)

\( \text{eval} = \text{nf} \ . \text{evalT} \)

Using this evaluation function, the expression \( \text{fst} (3, \text{snd} 5) \) now evaluates to the expected value 3.

Full non-strict evaluation, however, is only one option that we now have. We can stipulate additional strictness if desired, similarly to Haskell’s strictness annotations. The following function makes every constructor strict by evaluating each of its arguments to \( \text{whnf} \):

\( \text{strict} :: (f \sim g, \text{Traversable } f, \text{Monad } m, \ldots) \Rightarrow f (\text{Term} (m \oplus g)) \rightarrow \text{Term} (m \oplus g) \)

\( \text{strict} = \text{thunk} \ . \text{liftM} \ \text{inject} \ . \text{mapM} \ (\text{liftM} \ \text{inject} \ . \text{whnf}) \)

Now we can, for example, make all value constructors strict simply by replacing \( \text{inject} \) with \( \text{strict} \):

\( \text{instance} \ \text{EvalT Value where} \)

\( \text{evalAlgT} = \text{strict} \)

We can even be more specific: It is possible to define the following combinator, which takes a specification of which arguments are supposed to be strict and then performs the desired evaluation strategy:

\( \text{strictAt} :: (f \sim g, \text{Traversable } f, \text{Monad } m, \ldots) \Rightarrow
\forall a \ . \text{Ord } a \Rightarrow f a \rightarrow [a] \Rightarrow f (\text{Term} (m \oplus g)) \rightarrow \text{Term} (m \oplus g) \)

For example, we can make only the second component of the \( \text{Pair} \) value constructor strict:

\( \text{instance} \ \text{EvalT Value where} \)

\( \text{evalAlgT} = \text{strictAt spec} \)

\( \text{where} \)

\( \text{spec} \ (\text{Pair } a b) = [b] \)

\( \text{spec } _\_ = [] \)

In a similar manner also other combinators can be formed that allow to specify the evaluation strategy in a very fine grained fashion.

4 Conclusions

This simple observation shows yet another useful aspect of using compositional data types as a framework for dealing with abstract syntax trees \([1,2]\).

In addition to the example presented here, we have applied similar techniques to also control the evaluation strategy for other recursion schemes such as tree homomorphisms, tree transducers and attribute grammars.

References