# Evaluation à la carte

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):*
Bahr, P. (2011). Evaluation à la carte: non-strict evaluation via compositional data types. In P. Pettersson, & C. Seceleanu (Eds.), *Proceedings of the 23rd Nordic Workshop on Programming Theory* (pp. 38-40). Mälardalen University. Technical Report, Vol.. 254

# 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 tasks when writing compilers and interpreters.

As an example, consider the following *Haskell* [4] definition of an algebraic data type representing a simple expression language over integers and pairs:

$$\begin{aligned}&\textbf{data}\ \mathit{Exp} = \mathit{Const}\ \mathit{Int} \mid \mathit{Pair}\ \mathit{Exp}\ \mathit{Exp}\\ &\qquad\quad \mid \mathit{Add}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Fst}\ \mathit{Exp} \mid \mathit{Snd}\ \mathit{Exp}\end{aligned}$$

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:

$$\begin{aligned}&\mathit{eval} :: \mathit{Exp} \to \mathit{Exp}\\ &\mathit{eval}\ (\mathit{Const}\ i) = \mathit{Const}\ i\\ &\mathit{eval}\ (\mathit{Pair}\ x\ y) = \mathit{Pair}\ (\mathit{eval}\ x)\ (\mathit{eval}\ y)\\ &\mathit{eval}\ (\mathit{Add}\ x\ y) = \textbf{case}\ (\mathit{eval}\ x, \mathit{eval}\ y)\ \textbf{of}\\ &\quad (\mathit{Const}\ i, \mathit{Const}\ j) \to \mathit{Const}\ (i+j)\\ &\mathit{eval}\ (\mathit{Fst}\ p) \quad = \textbf{case}\ \mathit{eval}\ p\ \textbf{of}\ \mathit{Pair}\ x\ y \to x\\ &\mathit{eval}\ (\mathit{Snd}\ p) \quad = \textbf{case}\ \mathit{eval}\ p\ \textbf{of}\ \mathit{Pair}\ x\ y \to y\end{aligned}$$

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

$$\textbf{data}\ \mathit{Term}\ f = \mathit{Term}\ f\ (\mathit{Term}\ f)$$

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:

$$\textbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ e) \mid \mathit{Inr}\ (g\ e)$$

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

$$\begin{aligned}&\textbf{data}\ \mathit{Value}\ e = \mathit{Const}\ \mathit{Int} \mid \mathit{Pair}\ e\ e\\ &\textbf{data}\ \mathit{Op}\ e \quad\ = \mathit{Add}\ e\ e \mid \mathit{Fst}\ e \mid \mathit{Snd}\ e\\ &\textbf{type}\ \mathit{Sig} \qquad = \mathit{Op} \oplus \mathit{Value}\end{aligned}$$

This allows us to represent values and expressions as *Term Value* and *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:

$$inject :: (f \prec g) \Rightarrow f\ (Term\ g) \rightarrow Term\ g$$

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

$$\begin{array}{l}\textbf{class}\ Eval\ f\ \textbf{where}\\ \quad evalAlg :: f\ (Term\ Value) \rightarrow Term\ Value\end{array}$$

The instantiation of this class for values is trivial:

$$\begin{array}{l}\textbf{instance}\ Eval\ Value\ \textbf{where}\\ \quad evalAlg = inject\end{array}$$

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

$$\begin{array}{l}\textbf{instance}\ Eval\ Op\ \textbf{where}\\ \quad evalAlg\ (Add\ x\ y) = \textbf{case}\ (x,y)\ \textbf{of}\\ \qquad (Term\ (Const\ i), Term\ (Const\ j))\\ \qquad\quad \rightarrow inject\ (Const\ (i+j))\\ \quad evalAlg\ (Fst\ p) \quad = \textbf{case}\ p\ \textbf{of}\\ \qquad Term\ (Pair\ x\ y) \rightarrow x\\ \quad evalAlg\ (Snd\ p) \quad = \textbf{case}\ p\ \textbf{of}\\ \qquad Term\ (Pair\ x\ y) \rightarrow y\end{array}$$

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. *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 [3, 1], i.e. a function of type $f\ r \rightarrow m\ r$ for a monad $m$. Such a monadic algebra gives rise to a monadic catamorphism of type $Term\ f \rightarrow m\ r$. The evaluation algebra from Section 2 can be easily adapted to such a monadic style. Unfortunately, this will determine the evaluation strategy: The arguments of the operator symbols such as *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 $fst\ (3, snd\ 5)$ will yield an error since the evaluation of $(3, snd\ 5)$ 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 $m\ (Term\ Value)$, we therefore use the result type $Term\ (m \oplus Value)$ – making the monad part of the target signature. A monadic computation can thus be embedded into the term structure.

$$\begin{array}{l}thunk :: m\ (Term\ (m \oplus f)) \rightarrow Term\ (m \oplus f)\\ thunk = inject\end{array}$$

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

$$\begin{array}{l}whnf :: Monad\ m \Rightarrow\\ \qquad\quad Term\ (m \oplus f) \rightarrow m\ (f\ (Term\ (m \oplus f)))\\ whnf\ (Term\ (Inl\ m)) = m \gg\!= whnf\\ whnf\ (Term\ (Inr\ t)) = return\ t\end{array}$$

We can now use this idea to define a non-strict monadic evaluation function using the *Maybe* monad to indicate failure:

$$\begin{array}{l}\textbf{class}\ EvalT\ f\ \textbf{where}\\ \quad evalAlgT\ ::\ f\ (Term\ (Maybe \oplus Value))\\ \qquad\qquad\quad \rightarrow Term\ (Maybe \oplus Value)\end{array}$$

Again, the case for the value constructors is trivial:

$$\begin{array}{l}\textbf{instance}\ EvalT\ Value\ \textbf{where}\\ \quad evalAlgT = inject\end{array}$$

For evaluating the operator symbol applications, we simply evaluate their arguments to whnf and create a thunk in the end:

$$\begin{array}{l}evalAlgT\ (Add\ x\ y) = thunk\ \$\ \textbf{do}\\ \quad Const\ i \leftarrow whnf\ x\\ \quad Const\ j \leftarrow whnf\ y\end{array}$$

$$\begin{array}{l}
\quad \mathit{return}\ (\mathit{inject}\ (\mathit{Const}\ (i+j)))\\
\mathit{evalAlgT}\ (\mathit{Fst}\ v) \quad = \mathit{thunk}\,\$\,\mathbf{do}\\
\quad \mathit{Pair}\ x\ y \leftarrow \mathit{whnf}\ v\\
\quad \mathit{return}\ x\\
\mathit{evalAlgT}\ (\mathit{Snd}\ v) \quad = \mathit{thunk}\,\$\,\mathbf{do}\\
\quad \mathit{Pair}\ x\ y \leftarrow \mathit{whnf}\ v\\
\quad \mathit{return}\ y
\end{array}$$

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

$$\begin{array}{l}
\mathit{evalT} :: \mathit{Term}\ \mathit{Sig} \to \mathit{Term}\ (\mathit{Maybe} \oplus \mathit{Value})\\
\mathit{evalT} = \mathit{cata}\ \mathit{evalAlgT}
\end{array}$$

With only mild assumptions on the signature functors, we can also easily implement the evaluation to normal form by simply iterating the *whnf* function:

$$\begin{array}{l}
\mathit{nf} :: (\mathit{Monad}\ m, \mathit{Traversable}\ f) \Rightarrow\\
\qquad \mathit{Term}\ (m \oplus f) \to m\ (\mathit{Term}\ f)\\
\mathit{nf} = \mathit{liftM}\ \mathit{Term} \,.\, \mathit{mapM}\ \mathit{nf} \lll \mathit{whnf}
\end{array}$$

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

$$\begin{array}{l}
\mathit{eval} :: \mathit{Term}\ \mathit{Sig} \to \mathit{Maybe}\ (\mathit{Term}\ \mathit{Value})\\
\mathit{eval} = \mathit{nf} \,.\, \mathit{evalT}
\end{array}$$

Using this evaluation function, the expression $\mathit{fst}\ (3, \mathit{snd}\ 5)$ 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 evealuating each of its arguments to whnf:

$$\begin{array}{l}
\mathit{strict} :: (f \prec g, \mathit{Traversable}\ f, \mathit{Monad}\ m) \Rightarrow\\
\qquad f\ (\mathit{Term}\ (m \oplus g)) \to \mathit{Term}\ (m \oplus g)\\
\mathit{strict} = \mathit{thunk} \,.\, \mathit{liftM}\ \mathit{inject}\ .\\
\qquad\qquad \mathit{mapM}\ (\mathit{liftM}\ \mathit{inject} \,.\, \mathit{whnf})
\end{array}$$

Now we can, for example, make all value constructors strict simply by replacing *inject* with *strict*:

$$\begin{array}{l}
\mathbf{instance}\ \mathit{EvalT}\ \mathit{Value}\ \mathbf{where}\\
\quad \mathit{evalAlgT} = \mathit{strict}
\end{array}$$

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:

$$\begin{array}{l}
\mathit{strictAt} :: (f \prec g, \mathit{Traversable}\ f, \mathit{Monad}\ m, \ldots) \Rightarrow\\
\qquad (\forall\ a \,.\, \mathit{Ord}\ a \Rightarrow f\ a \to [a]) \to\\
\qquad f\ (\mathit{Term}\ (m \oplus g)) \to \mathit{Term}\ (m \oplus g)
\end{array}$$

For example, we can make only the second component of the *Pair* value constructor strict:

$$\begin{array}{l}
\mathbf{instance}\ \mathit{EvalT}\ \mathit{Value}\ \mathbf{where}\\
\quad \mathit{evalAlgT} = \mathit{strictAt}\ \mathit{spec}\\
\qquad \mathbf{where}\ \mathit{spec}\ (\mathit{Pair}\ a\ b) = [b]\\
\qquad\qquad \mathit{spec}\ _ \qquad\qquad = [\,]
\end{array}$$

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

[1] Patrick Bahr and Tom Hvitved. Compositional data types. WGP 2011, to appear.

[2] Laurence Day and Graham Hutton. Towards Modular Compilers For Effects. In *Proceedings of the Symposium on Trends in Functional Programming*, Madrid, Spain, 2011.

[3] Maarten Fokkinga. Monadic Maps and Folds for Arbitrary Datatypes. Technical report, Memoranda Informatica, University of Twente, 1994.

[4] Simon Marlow. Haskell 2010 Language Report, 2010.

[5] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.