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):
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.

As an example, consider the following Haskell [4] 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:

\[
\begin{align*}
eval (\text{Fst } p) &= \text{case } eval\ p\ of\ \text{Pair } x y \to x \\
eval (\text{Snd } p) &= \text{case } eval\ p\ of\ \text{Pair } x y \to y
\end{align*}
\]

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:

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 = Inl (f e) | 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 Term Value and Term Sig, respectively.

In addition, Swierstra also defines a binary type class \( \triangleleft \) on signature functors that approximates inclusion. That is, \( f \triangleleft g \) 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:

\[ \text{inject} :: (f \prec g) \Rightarrow f \text{ (Term } g\text{)} \rightarrow \text{Term } g \]

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

\[
\begin{align*}
\text{class} & \quad \text{Eval } f \\
\text{evalAlg} & :: f \text{ (Term Value)} \rightarrow \text{Term Value}
\end{align*}
\]

The instantiation of this class for values is trivial:

\[
\begin{align*}
\text{instance} & \quad \text{Eval Value} \\
\text{evalAlg} & = \text{inject}
\end{align*}
\]

For operator symbols we have to provide an im-
plementation that evaluates the arguments appro-
priately:

\[
\begin{align*}
\text{instance} & \quad \text{Eval Op} \\
\text{evalAlg} & (\text{Add } x \ y) = \text{case } (x, y) \text{ of} \\
& \quad \text{Term } (\text{Const } i), \text{Term } (\text{Const } j) \\
& \quad \rightarrow \text{inject } (\text{Const } (i + j)) \\
\text{evalAlg} & (\text{Fst } p) = \text{case } p \text{ of} \\
& \quad \text{Term } (\text{Pair } x \ y) \rightarrow x \\
\text{evalAlg} & (\text{Snd } p) = \text{case } p \text{ of} \\
& \quad \text{Term } (\text{Pair } x \ y) \rightarrow y
\end{align*}
\]

Note that the case distinction in the above eval-
uation algebra as well as in the direct evaluation in
Section 2 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 al-
gebra \[3\] \[1\], i.e. a function of type \( f \rightarrow m r \) for
a monad \( m \). Such a monadic algebra gives rise
to a monadic catamorphism of type \( \text{Term } f \rightarrow m r \).
The evaluation algebra from Section 2 can be easily
adapted to such a monadic style. Unfortunately,

\[
\begin{align*}
\text{thunk} & :: m \text{ (Term } (m \oplus f)) \rightarrow \text{Term } (m \oplus f) \\
\text{thunk} & = \text{inject}
\end{align*}
\]

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

\[
\begin{align*}
\text{whnf} & :: \text{Monad } m \Rightarrow \\
& \quad \text{Term } (m \oplus f) \rightarrow m (f \text{ (Term } (m \oplus f))) \\
\text{whnf} & (\text{Term } (\text{Inl } m)) = m \gg= \text{whnf} \\
\text{whnf} & (\text{Term } (\text{Inr } t)) = \text{return } t
\end{align*}
\]

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

\[
\begin{align*}
\text{class} & \quad \text{EvalT } f \\
\text{evalAlgT} & :: f \text{ (Term } (\text{Maybe } \oplus \text{Value})) \\
& \quad \rightarrow \text{Term } (\text{Maybe } \oplus \text{Value})
\end{align*}
\]

Again, the case for the value constructors is trivial:

\[
\begin{align*}
\text{instance} & \quad \text{EvalT Value} \\
\text{evalAlgT} & = \text{inject}
\end{align*}
\]

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

\[
\begin{align*}
\text{evalAlgT} & (\text{Add } x \ y) = \text{thunk } \$ \text{ do} \\
& \quad \text{Const } i \leftarrow \text{whnf } x \\
& \quad \text{Const } j \leftarrow \text{whnf } y
\end{align*}
\]
return (inject (Const (i+j)))

\[
\text{evalAlgT (Fst v) } = \text{thunk}\ \
\text{do}\ \
\text{Pair x y } \leftarrow \text{whnf v}\ \
\text{return x}
\]

\[
\text{evalAlgT (Snd v) } = \text{thunk}\ \
\text{do}\ \
\text{Pair x y } \leftarrow \text{whnf v}\ \
\text{return y}
\]

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

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

\[
\text{evalT } = \text{cata evalAlgT}
\]

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:

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

\[
\text{nf } = \text{liftM Term . mapM nf } \ll\ll \text{whnf}
\]

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

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

\[
\text{eval } = \text{nf . evalT}
\]

Using this evaluation function, the expression \text{fst (3, 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 eevaluating each of its arguments to \text{whnf}:

\[
\text{strict :: (f } \times g, \text{Traversable f, Monad m)} \Rightarrow \text{f (Term (m } \oplus \text{g)} } \rightarrow \text{Term (m } \oplus \text{g)}
\]

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

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

\[
\text{instance EvalT Value where}
\]

\[
\text{evalAlgT } = \text{strict}
\]

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:

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

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

\[
\text{instance EvalT Value where}
\]

\[
\text{evalAlgT } = \text{strictAt spec}
\]

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

\[
\text{spec _ } = [1]
\]

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


