A Lightweight Optimization Technique for Data Types à la Carte

Hirotada Kiriyama Tomoyuki Aotani Hidehiko Masuhara
Tokyo Institute of Technology, Japan
kiriyama.h.ab@m.titech.ac.jp aotani@is.titech.ac.jp masuhara@acm.org

Abstract

Data types à la carte (DTC) is a technique for adding new variants to data types modularly. A drawback of DTC compared with simple variant types, which are commonly used to define data types in functional programming languages, is runtime inefficiency caused by the destruction of these values.

In this paper, we propose a lightweight optimization technique for functions that destruct the values of DTC data types. It makes their execution as efficient as their non-DTC counterparts by just (1) deriving non-extensible algebraic data types isomorphic to extensible data types defined in DTC fashion and (2) using them within the type annotations that specify concrete data types using the composition operator given in DTC. The approach is based on an insight on the functions: the functions never depend on any concrete data types but merely constrain them.

We implemented functions that take an extensible data type defined in DTC fashion and derive an isomorphic non-extensible algebraic data type using Template Haskell. Our experimental results show that DTC functions using our approach run as efficiently as their non-DTC counterparts and avoid performance slow down even if the data types are extended multiple times.

General Terms Language,Performance

Keywords Expression problem, modularity, reusability

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Data Types and Structures

1. Introduction

Data types à la carte (DTC) [5] is a technique for achieving modularly extensible recursive data types. It can be seen as a solution to the expression problem [6]: "The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety (e.g., no casts)." We call programs written in DTC fashion *DTC programs* in this paper.

From the data type definition point of view, DTC extends the idea of two-level types[4] with coproduct and fixed point operators, namely $\oplus$ and `Mu` respectively, over structure operators. The coproduct operator $\oplus$ builds a structure operator by composing two structure operators. We call the resulting structure operators *composed structure operators* and otherwise use the term *non-composed structure operator*. The fixed point operator `Mu` takes a structure operator, say `F`, and creates a recursive data type by feeding `Mu F` to `F`. `List` and `Maybe` in Haskell are two examples of non-composed structure operators.

Functions over such recursive data types are defined in DTC using ad-hoc polymorphism and catamorphism. Suppose `F` and `G` are structure operators. If function `f` is defined over `F a` and `G a`, `f` is also defined over `(F⊕G) a` in DTC. Catamorphism lifts function `f` over `F a` to `Mu F`. A function lifted by catamorphism `(f :: F a → a)` is called *algebra*.

DTC also provides a binary constraint $\prec$ over structure operators. `F` $\prec$ `G` intuitively means that for all type `a`, values of type `F a` can be used as values of type `G a`. A simple example is taking the functor `G` as `H⊕F` in a relation of `F` $\prec$ `G` where `H` is some structure operator. A binary constraint is used to define *smart constructors*, which are functions that build values polymorphic over structure operators. For example, the type of smart constructor `f` that just returns a value built with data constructor $\mathtt{F_C}$ of structure operator `F` is `s a` rather than `F a` for any structure operator `s` satisfying `F`$\prec$`s`.

Notably, DTC is ready to be used to develop extensible programs in today's Haskell with respect to required language mechanisms. In fact, we have (multi parameter) type classes for ad-hoc polymorphism and type constraints for declaring types of smart constructors. Moreover, there is a library [1] that generates the necessary boilerplate code for DTC programs at compile time using Template Haskell.

DTC is, however, not ready for practical use because it makes programs inefficient with respect to runtime performance compared with their non-DTC counterparts. This is because DTC forces us to compose structure operators merely linearly as type-level lists. It therefore takes linear time to decompose a value of a composed structure operator, i.e., $O(n)$ where n is the number of structure operators composed with $\oplus$.

We propose a lightweight optimization technique for DTC programs as a solution to the problem. It makes DTC programs run as efficiently as their non-DTC counterparts by merely (1) deriving one non-composed structure operator isomorphic to the composed structure operator from a given composed structure operator, and (2) using it within the type annotations on functions. This technique reflects our insight that functions and values polymorphic over structure operators never specify concrete structure operators. Concrete structure operators merely appear in their type constraints instead. In other words, our technique never changes the way to define functions and values polymorphic over structure operators in DTC.

MODULARITY Companion'16, March 14–17, 2016, Málaga, Spain
ACM. 978-1-4503-4033-5/16/03...$15.00
http://dx.doi.org/10.1145/2892664.2892677

```
data BaseExpF e = Lit Int | Add e e deriving Functor
data Mu f = Mu {unwrap :: f (Mu f)}

eval :: (Eval f, Functor f) ⇒ Mu f → Int
eval = fold evalAlgebra

fold :: Functor f ⇒ (f a → a) → Mu f → a
fold f = run
  where run = f ∘ (fmap run) ∘ unwrap

class Eval f where
  evalAlgebra :: f Int → Int

instance Eval BaseExpF where
  evalAlgebra (Lit i) = i
  evalAlgebra (Add x y) = x + y
```

Figure 1: Simple example of DTC

Using Template Haskell, we implement a code generator that derives (1) non-composed structure operators from given composed structure operators and (2) instances of type classes necessary to use the non-composed structure operators instead of composed structure operators. Using these functions, programmers do not have to make additional efforts to make DTC programs efficient.

The rest of the paper is organized as follows. Section 2 is a brief introduction of DTC using a simple interpreter as an example. In Section 3, we discuss the problem of DTC programs with respect to runtime efficiency and show our solution to the problem. In Section 5, we explain our implementation of the functions deriving non-composed structure operators from given composed structure operators using Template Haskell. In Section 6, we show the performance improvements achieved by our approach. In Section 7, we discuss related work, and Section 8 concludes the paper.

2. Data Types à la Carte

In this section, we review DTC using an interpreter for simple arithmetic expressions as an example. Suppose that the initial version of our interpreter evaluates an expression that consists of integer literals and the summation operator, and the second version adds support for the inversion operator additionally as the syntax of the expressions.

2.1 Initial Interpreter

Figure 1 shows an implementation of the abstract syntax and the evaluation function `eval` in DTC. `BaseExpF` (line 1) is a functor that implements the abstract syntax of our initial interpreter. The type parameter `e` is the type of subexpressions used in the summation operator. `Mu f` (line 2) is a fixpoint of functor `f`. Note that `Mu BaseExpF` is isomorphic to the standard implementation of the abstract syntax:

```
data BaseExp = Lit Int | Add BaseExp BaseExp
```

`eval` recursively applies `evalAlgebra` to subexpressions of the argument expression. `evalAlgebra` is polymorphic over functor `f`. It takes an expression of type `f Int` and returns an integer, where `f` is a functor. Lines 14 – 16 define `evalAlgebra` in the case that the functor is `BaseExpF`. For example, we get 7 by evaluating the following program that represents the arithmetic expression $3 + 4$.

```
eval $ Mu $ Add (Mu $ Lit 3) (Mu $ Lit 4)
```

2.2 Extending the Interpreter

It is possible in DTC to extend the syntax of the expression in our interpreter with the inversion operator without modifying `BaseExpF`. All programmers have to do is to define a new functor that implements the new part of the extended abstract syntax, and to define `evalAlgebra` for the case that the functor is `InvExpF` and `f⊕g` for any functor `f` and `g` such that `evalAlgebra` is defined, where $\oplus$ is the summation operator over functors defined as follows:

```
data InvExpF e = Inv e deriving Functor

instance Eval InvExpF where
  evalAlgebra (Inv i) = -i

instance (Eval f, Eval g) ⇒ Eval (f ⊕ g) where
  evalAlgebra (InL x) = evalAlgebra x
  evalAlgebra (InR x) = evalAlgebra x
```

Figure 2: Simple example of DTC

```
class sub ≺ sup where
  inj :: sub a → sup a
instance f ≺ f where
  inj = id
instance f ≺ (f ⊕ g) where
  inj = InL
instance (f ≺ g) ⇒ f ≺ (h ⊕ g) where
  inj = InR ∘ inj

inject :: (f ≺ g) ⇒ f (Mu g) → Mu g
inject = Mu ∘ inj
```

Figure 3: Definition of injection relations

```
lit :: (BaseExpF ≺ f) ⇒ Int → Mu f
lit = inject ∘ Lit
add :: (BaseExpF ≺ f) ⇒ Mu f → Mu f → Mu f
add x y = inject (Add x y)
inv :: (InvExpF ≺ f) ⇒ Mu f → Mu f
inv = inject ∘ Inv
```

Figure 4: Smart constructors

```
infixr ⊕
data (f ⊕ g) e = InL (f e) | InR (g e)
```

Figure 2 shows the code for the extension. Functor `InvExpF` implements only the new part of the extend abstract syntax and thus `evalAlgebra` for `InvExpF` handles only one case.

The data type that implements the fully extended abstract syntax is `Mu (BaseExpF⊕InvExpF)`. Therefore, we need to define `evalAlgebra` for functor `BaseExpF⊕InvF`. We define it more generally in the last instance declaration of Figure 2.

Note that `eval` is polymorphic over functor `f` and is thus applicable to expressions that consist of `Lit`, `Add` and `Inv`. For example, we get 2 by evaluating the following program.

```
eval $ Mu $ InL $ Add (Mu $ InL $ Add (Mu $ InL $ Lit 3)
                                      (Mu $ InL $ Lit 4))
                      (Mu $ InR $ Inv $ Mu $ InL $ Lit 5)
```

2.3 Constructing Reusable Expressions

It is also possible in DTC to build expressions that are reusable in extended interpreters via *smart constructors*, which allow us to avoid using data constructors `Mu`, `InL`, and `InR` explicitly.

```
data Exp' = Add Exp' Exp' | Inv Exp' | Lit Int
eval' :: Exp' → Int
eval' (Add' e1 e2) = eval' e1 + eval' e2
eval' (Inv' e)     = - eval' e
eval' (Lit' i)     = i
```

Figure 5: Definitions of non-composed structure operator and eval'

Intuitively, smart constructors automatically apply `Mu` only once and `InL` and `InR` as many times as necessary according to the types of the constructed data. For example, the smart constructor `lit` takes an integer as `Lit` and applies `Lit` and `Mu` to the integer if the desired type of the constructed data is `Mu BaseExpF`, while it applies `Lit`, `InL` and `Mu` once if the type is `Mu (BaseExpF⊕InvExpF)`.

To this end, DTC uses the injection relation $f \prec g$ among the functors defined in Figure 3. `inj` injects the data of type `f a` into `(f⊕h) a` and `(h⊕f) a` by applying `InL` and `InR`, respectively, for any functor `h`. The auxiliary function `inject` applies `Mu` after applying `InL` and `InR` via `inj`.

Figure 4 shows smart constructors `lit`, `add` and `inv`. They merely apply `inject` to the data constructed by `Lit`, `Add` and `Inv`. Note that they are polymorphic over the functor `f` that satisfies each constraints. For example, `lit` builds expressions of type `Mu f` if we can get the data of type `f a` by applying only `InL` and `InR` zero or more times to the data of type `BaseExpF a`.

Expressions built via smart constructors are polymorphic over functors. For example, we can define an expression that consists of integer literals and the summation operator and use it to build larger expressions that contain the inversion operator as follows.

```
exp1 :: (BaseExpF ≺ f) ⇒ Mu f
exp1 = lit 3 `add` lit 4

exp2 :: (InvExpF ≺ f, BaseExpF ≺ f) ⇒ Mu f
exp2 = exp1 `add` inv exp1
```

It is important to notice that expressions need type annotations when they are passed to `eval` as follows.

```
eval (exp2 :: Mu (BaseExpF ⊕ InvExpF))
```

DTC programs are extensible if we do not specify a concrete data type. This is also the key point of our approach to the performance problem.

3. Problem of DTC

One of the problems of using composed structure operators is that they make programs run slower than those that use only non-composed structure operators. This is because DTC programs compose several data constructors using `InL` and `InR` repeatedly. In other words, it is not possible for DTC programs to have one big `eval` function that deals with all the cases of expression without losing generality. Several steps are therefore necessary to reach the `Lit`, `Add` and `Inv` cases. For example, when `eval` is applied to `exp2` as in Section 2, `evalAlgebra` is applied 16 times, half of which are applied to `InL x` and `InR x`. If we use non-composed structure operators and `eval'` (Figure 5), `eval'` is applied only six times. The more functors we compose, the more significant the performance issue becomes. For example, a term representing a math expression using DTC can be written as the following form.

```
exp1 :: (f ≺ BaseExp) ⇒ Fix f
exp1 = add (lit 1) (lit 3)
```

`exp1` is constructed by smart constructors. When the type `f` is instantiated to $F_1 \oplus$ `BaseExpF`, the actual representation of `exp1` gets the following form.

```
data BaseInvF e = Lit' Int | Add' e e | Inv' e
                deriving Functor

instance Eval BaseInvF where
  evalAlgebra (Lit' x) = x
  evalAlgebra (Add' x y) = x + y
  evalAlgebra (Inv' x) = (- x)

instance BaseExpF ≺ BaseInvF where
  inj (Lit x) = Lit' x
  inj (Add x y) = Add' x y

instance InvExpF ≺ BaseInvF where
  inj (Inv x) = Inv' x
```

Figure 6: Compiled data types

```
exp1 :: Fix (F1 ⊕ BaseExpF)
exp1 = Fix (InR (Add (Fix (InR (Lit 1)))
                (Fix (InR (Lit 3)))))
```

When extending this functor with F_2, the number of constructors of `exp1` increases.

```
exp1 :: Fix (F2 ⊕ F1 ⊕ BaseExpF)
≡ Fix (InR (InR (Add (Fix (InR (InR (Lit 1)))))
                     (Fix (InR (InR (Lit 3)))))))
```

We are faced with two problems. First, because applications of not only `Lit`, `Add` and `Inv` but also `InR` and `InL` allocate memory, the program uses more memory than non-composed counterparts. More memory is allocated and thus more GC time required. Second, the number of function applications is larger than in their non-composed counterparts.

It could be a solution to reduce the number of applications of `evalAlgebra` by composing functors as binary trees using type families instead of type classes to define the injection relation $\prec$ among functors[1]. This however still causes significant slow down when we compose many functors compared to non-composed counterparts as we show in Section 6.

4. Our Approach

We solve the problem of DTC by changing data and function implementations at compile time without any modification to an existing program using DTC.

Our approach is fairly simple. It derives a functor F' from $\bigoplus F_i$. Functor F' is defined as a non-composed structure operator and provides all the data constructors available in F_i. It also derives instances of the type classes of which $\bigoplus F_i$ is an instance. For example, our approach derives functor `BaseInvF` from `BaseExpF⊕InvExpF` and instances of `Eval` and $\prec$ as shown in Figure 6. `Lit'`, `Add'` and `Inv'` are data constructors corresponding to `Lit`, `Add` and `Inv`. `evalAlgebra` for `BaseInvF` is defined using `evalAlgebra` for `BaseExpF` and `InvExpF`. The injection relations are derived from `BaseExpF` to `BaseInvF` and from `InvExpF` to `BaseInvF`.

One could derive another data type namely `BaseInvExp` that does not have type parameters as follows.

```
data BaseInvExp = Lit'' Int
                | Add'' BaseInvExp BaseInvExp
                | Inv'' BaseInvExp
```

This approach has, however, a significant disadvantage in that we can neither apply `eval` to the data built by `Lit''`, `Add''`, and `Inv''` nor use smart constructors to build them. In other words, we need to define a specialized version of `eval`, namely `eval'`, and

replace every `eval` call with `eval'`. We also have to use the data constructors `Lit''`, `Add''` and `Inv''` directly instead of smart constructors to build expressions. Effectively, we have to change the entire program.

Our approach, in comparison, does not require any changes to extensible parts of the programs. For example, we can build an expression of type `Mu BaseInvF` using the smart constructors in Figure 4 and evaluate it using `eval` in Figure 1 as follows.

```
eval (exp2 :: Mu BaseInvF)
```

Compared with the previous example in Section 2, only the type annotation that specifies the concrete functor is different. Because extensible programs do not specify any concrete functor, we can use it without any modifications.

5. Automated Derivation

In this section, we explain the automatic compile time derivation of non-composed structure operators and instances of `Eval` and $\prec$. As shown in Figure 6, the derivation is straightforward and therefore, it is desirable to automate it.

We explain our derivation algorithm as a compile-time meta-level function `compType`, which we can implement straightforwardly using Template Haskell.

5.1 Overview of the Deriving Process

`compType` takes three parameters necessary for derivations. The first and second parameters are the names of the derived non-composed structure operators and the composed structure operators, respectively. The last one is the names of the type classes of which the defined structure operators must be the instances.

For example, `compType` derives `BaseInvF` and the instance of `Eval` in Figure 6 from `InvF` $\oplus$ `BaseF` for the following code:

```
type ExpF = InvF ⊕ BaseF
compType "BaseInvF" ''ExpF ['']Eval]
```

It also derives the instances of $\prec$, i.e, `InvF` $\prec$ `BaseInvF` and `BaseF` $\prec$ `BaseInvF`.

The deriving process consists of three steps:

1. Deriving new datatype
2. Deriving injection (smart constructor)
3. Deriving algebra instances of the specified type class

The first step is deriving the non-composed structure operator. In this step, the deriving process receives the name of the composed structure operator, and generates the declaration of the non-composed structure operators and the mapping from a data constructor to a non-composed structure operator as internal information passed to the second and the third step.

The second step is deriving the injection `f` $\prec$ `g`. In this step, the deriving process receives the mapping between functors that is generated in step 1 and generates type class instances of subtype relations. This step enables the programmers to use smart constructors.

The third step is deriving typeclass instances for DTC function (called *algebra*). In this step, the deriving process requires algebra names and algebra declarations. and generates algebras for the composed structure operator.

5.2 Deriving a Non-composed Structure Operator

Let `G` be the non-composed structure operator derived from $\bigoplus F_i$. Then for each data constructor of F_i, there must be one unique data constructor of `G`. For example, `BaseInvF` has three data constructors, `Add'`, `Lit'` and `Inv'`, that correspond to the `Add` of `BaseF` and `Inv` of `InvF`, respectively.

In general, we can derive non-composed structure operators in the following way. Suppose that F_i is defined as follows.

$$\texttt{data } F_i\ a = Ci^1\ t_1^1\ \ldots\ t_{\rho i_1}^1 \mid Ci^2\ t_1^2\ \ldots\ t_{\rho i_2}^2 \mid \ldots \mid Ci^{ki}\ t_1^{ki}\ \ldots\ t_{\rho i_k}^{ki}$$

a is a type variable, and it can appear in each type t_p^q. Then, we define `G` by creating a new data constructor name Cn'^i for Cn as follows.

$$\begin{aligned}\texttt{data G a} = {} & C1'^1\ t1_1^1\ \ldots\ t1_{\rho 1_1}^1 \mid \ldots \mid C1'^{k1}\ t1_1^{k1}\ \ldots\ t1_{\rho 1_k}^{k1} \\ \mid {} & C2'^1\ t2_1^1\ \ldots\ t2_{\rho 2_1}^1 \mid \ldots \mid C2'^{k2}\ t2_1^{k2}\ \ldots\ t2_{\rho 2_k}^{k2} \\ & \ldots \\ \mid {} & Cn'^1\ tn_1^1\ \ldots\ tn_{\rho n_1}^1 \mid \ldots \mid Cn'^{kn}\ tn_1^{kn}\ \ldots\ tn_{\rho n_k}^{kn}\end{aligned}$$

5.3 Deriving Injective Relation

To use the non-composed structure operator `G` derived from $\bigoplus F_i$ instead of $\bigoplus F_i$, it is necessary to ensure that we can use `G` whenever F_i and any of their compositions are used. This is done by simply defining the injection relation $F_i \prec$ `G` for each F_i as follows:

$$\begin{aligned}&\texttt{instance } F_i \prec \texttt{G where} \\ &\quad \texttt{inj x = case x of} \\ &\quad\quad Ci^1\ t_1^1\ \ldots\ t_{\rho i_1}^1 \rightarrow Ci'^1\ t_1^1\ \ldots\ t_{\rho i_1}^1 \\ &\quad\quad \ldots \\ &\quad\quad Ci^{ki}\ t_1^{ki}\ \ldots\ t_{\rho i_k}^{ki} \rightarrow Ci'^{ki}\ t_1^{ki}\ \ldots\ t_{\rho i_k}^{ki}\end{aligned}$$

We do not need to define the injection relations from compositions of F_i to `G`. This is because compositions of F_i never appear in the types of extensible functions. For example, the type of `exp3` in Section 3 specifies that there are two injection relations from `BaseExpF` to `f` (`BaseExpF` $\prec$ `f`) and from `InvExpF` to `f` (`InvExpF` $\prec$ `f`), instead of specifing the injection relation from (`BaseExpF` $\oplus$ `InvExpF`) to `f` and (`InvExpF` $\oplus$ `BaseExpF`) to `f`.

It is not good to specify (`BaseExpF` $\oplus$ `InvExpF`) $\prec$ `f` because it specifies the order of the compositions of `BaseExpF` and `InvExpF` and thus limits the extensibility of the function.

5.4 Deriving Type Class Instances

```
instance A NewExpF where
  f x = case x of
    ...
    Ci a1 .. an → f (C'i a1 .. an)
    ...
```

Figure 7: Derived typeclass instance for algebra function

Let `A` be a typeclass that has the algebra function, `G` be a derived non-composed structure operator and `f` be a algebra function.

Figure 7 shows the derived instance declaration in this step. C_i is a data constructor of the derived non-composed operator and C'_i is the counterpart of the base operator. The derived function simply maps the C_i to C'_i and applies the base operator's algebra.

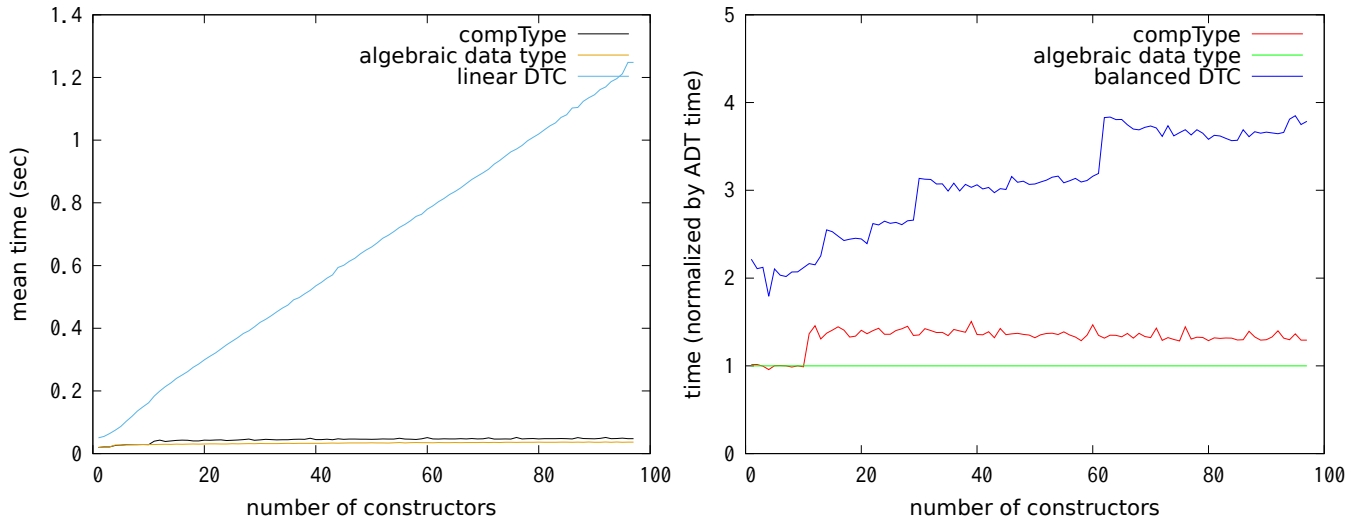
6. Evaluation

To measure the speed-up, we made a simple benchmark program and compared the optimized program with its non-optimized counterpart.

6.1 Benchmarks

Expression Evaluation

We built simple expression trees and measured the time taken to evaluate them. In this benchmark, generated trees are perfect binary trees whose heights are fixed to 20.



(a) Result of compType, original DTC and ADT

(b) Result of compType, balanced DTC and ADT

Figure 8: Mean running time relative to non-DTC program

```
data Val e = Val Int
data Add_i e = Add_i e e
type Sum_n = Val ⊕ Add_1 ⊕ ... ⊕ Add_n
instance Eval Val where
  evalAlgebra (Val x) = x
instance Eval Add_i where
  evalAlgebra (Add_i l r) = l + r
buildTree 0 = return $ Val 1
buildTree n = do
  k ← randomRIO (0,len)
  adds !! k <$> buildTree (n - 1) <*> buildTree (n - 1)
  where
    adds = [add1,add2,add3, .. , addn]
    len = length adds
$(compType ''Sum_n)
eval :: Eval f ⇒ Mu f → Int
eval = fold evalAlgebra
```

Figure 9: Code of expression evaluation benchmark

6.2 Results

The benchmarks were performed on a Linux machine running kernel version 4.0.1 on an Intel Core i7 3.3 GHz with 32 GB of RAM. Times were measured by the package *Criterion* [3] using GHC 7.10.1 with flag -O2.

6.2.1 Speedup in Data Decomposition

Figure 8a shows the relation between the number of composed constructors (functors) and evaluation time. When increasing the number of composition operators, the time required for the original DTC increases linearly. In contrast, the time of the optimized versions of DTC and algebraic data type (ADT) remained constant.

Performance normalized by ADT is shown in Figure 8b. This graph contains the result of the ADT, balanced DTC and DTC optimized by our approach. Our simple optimization technique reduces the decomposition from $O(N)$ to $O(1)$ where N is the number of functor sum operators $\oplus$.

We can see that DTC programs spend a time proportional to the number of constructors (functors). This result means that the average cost of applying algebras is $O(N)$. In contrast, composed functors and ADT require constant time even if the number of functors increases.

Figure 8b shows benchmark results of DTC composing functors with binary tree and compType normalized by ADT time.

7. Related Works

The idea of behind our mechanism is not new. The idea replacing the function with a more efficient function has been used in compiler implementation and the application for functional programming is known as the *worker/wrapper transformation* [2]. Our approach extends this idea to polymorphic and extensible functions.

8. Conclusion

We proposed a lightweight optimization technique for functions that destruct the values of DTC data types. This technique does not require any changes to the existing DTC programs except for their type annotations. We also implemented a code generator from composed type operators to non-composed structure operators using Template Haskell. Our experimental results show that a DTC program using our approach runs faster than the original DTC and avoids slowdown even if the data types are extended multiple times.

References

[1] P. Bahr. Composing and decomposing data types: A closed type families implementation of data types à la carte. In *Proceedings of WGP '14*, pages 71–82, 2014.

[2] A. Gill and G. Hutton. The worker/wrapper transformation. *J. Funct. Program.*, 19(2):227–251, Mar. 2009.

[3] B. OSullivan. Criterion version 1.1.0. `http://hackage.haskell.org/package/criterion`, 2015.

[4] T. Sheard and E. Pasalic. Two-level types and parameterized modules. *Journal of Functional Programming*, 14(5):547–587, 2004.

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

[6] P. Wadler and et al. The expression problem. Discussion on the Java-Genericity mailing list, December 1998.