Functional Pearl: A Smart View on Datatypes

Mauro Jaskelioff Exequiel Rivas
CIFASIS-CONICET, Argentina
Universidad Nacional de Rosario, Argentina
jaskelioff@cifasis-conicet.gov.ar

Abstract

Left-nested list concatenations, left-nested binds on the free monad, and left-nested choices in many non-determinism monads have an algorithmically bad performance. Can we solve this problem without losing the ability to pattern-match on the computation? Surprisingly, there is a deceptively simple solution: use a smart view to pattern-match on the datatype. We introduce the notion of smart view and show how it solves the problem of slow left-nested operations. In particular, we use the technique to obtain fast and simple implementations of lists, of free monads, and of two non-determinism monads.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.2 [*Programming Languages*]: Language Classifications – Applicative (functional) languages; E.1 [*Data Structures*]: Lists, stacks, and queues

Keywords List, Monad, MonadPlus, Data Structure

1. Introduction

Lists are one of the most important data structures in functional programming. However, the append operation $(+\!\!+)$ is inefficient, as it is linear on the first argument. Therefore, in a left-nested concatenation $((xs +\!\!+ ys) +\!\!+ zs)$ we are going to pay the price of traversing xs twice. A typical example of such a situation is the function $reverse$:

$$
\begin{aligned}
&reverse :: [a] \to [a] \\
&reverse\ [\,] \qquad\ = [\,] \\
&reverse\ (x : xs) = reverse\ xs +\!\!+ [x]
\end{aligned}
$$

Unfolding the recursion, $reverse\ [1, 2, 3, 4]$ amounts to

$$((([\,] +\!\!+ [4]) +\!\!+ [3]) +\!\!+ [2]) +\!\!+ [1].$$

Left-nested appends make this function quadratic on the length of the input list.

Rather than rewriting the function $reverse$ (which would only solve the problem for this particular function) we want a new data structure for lists that will make functions like $reverse$ fast. More precisely, we want a *catenable list*. That is, a data structure for lists that has fast appends and fast pattern-matching.

ICFP '15, September 01 – 03, 2015, Vancouver, BC, Canada.

ACM 978-1-4503-3669-7/15/09. . . $15.00.
http://dx.doi.org/10.1145/2784731.2784743

rivas@cifasis-conicet.gov.ar

The problem of optimising list concatenation is just one instance of a more general problem which occurs in other settings, such as libraries of effects based on free monads and implementations of domain specific languages. In this article we present an extremely simple technique for optimising selected operations using their algebraic properties while keeping the efficiency of pattern-matching. We achieve this by transforming a data structure into a new one, which is inspected using a *smart view*. We illustrate the technique with several examples: catenable lists (Section 2), free monads (Section 3), and two different implementations of non-determinism monads (Section 4).

The search of efficient list implementations and the generalisation of the ideas to other datatypes has a long history. We review related work in Section 5, and run some benchmarks in Section 6 that show that data structures with smart views are quite fast indeed.

We use Haskell to explain the ideas, but lazyness does not play any significant role. In fact, we also implement and benchmark smart views for lists in the strict language ML.

2. Catenable Lists

In this section we take the basic datatype of lists and transform it into a fast implementation of catenable lists.

2.1 Basic Lists

We define our own datatype of lists, rather than reuse the one predefined in Haskell, in order to be able to alter it.

$$\begin{aligned}&\mathbf{data}\ \mathit{List}\ a = \mathit{Nil}\\ &\qquad\qquad\quad \mid\ \mathit{Cons}\ a\ (\mathit{List}\ a)\end{aligned}$$

With this datatype we have constant time construction of the empty list (Nil), consing of an element (Cons), and pattern-matching. However, list concatenation is expensive as it is linear in its first argument:

$$\begin{aligned}&(+\!\!+) :: \mathit{List}\ a \to \mathit{List}\ a \to \mathit{List}\ a\\ &\mathit{Nil} \qquad\quad +\!\!+\ ys = ys\\ &\mathit{Cons}\ x\ xs +\!\!+\ ys = \mathit{Cons}\ x\ (xs +\!\!+\ ys)\end{aligned}$$

We are interested in obtaining a representation for lists with a fast implementation of concatenation. However, while some functions such as

$$\begin{aligned}&\mathit{wrap} :: a \to \mathit{List}\ a\\ &\mathit{wrap}\ x = \mathit{Cons}\ x\ \mathit{Nil}\end{aligned}$$

only use constructors, the most common way of defining functions on lists is by pattern-matching:

$reverse :: List\ a \to List\ a$
$reverse\ Nil \qquad\quad = Nil$
$reverse\ (Cons\ x\ xs) = reverse\ xs +\!\!+ wrap\ x$
Therefore, we do not want to lose the ability to pattern-match efficiently.

2.2 A Smart View on Lists

In order to get lists with fast concatenation, we add a constructor $(:+\!\!+)$ that represents this operation:

$$\begin{aligned}\mathbf{data}\ List\ a = {} & Nil \\ \mid {} & Cons\ a\ (List\ a) \\ \mid {} & List\ a :+\!\!+ List\ a\end{aligned}$$

Now concatenation has become cheap as it is simply the application of the constructor $(:+\!\!+)$. In order to be able to define functions by pattern-matching as before, we define a view [8]:

$$\mathbf{data}\ ListView\ a = Nil_V \mid Cons_V\ a\ (List\ a)$$

Given a function $view_L :: List\ a \to ListView\ a$, we can define $reverse$ as:

$$\begin{aligned}&reverse\ xs = \mathbf{case}\ view_L\ xs\ \mathbf{of} \\ &\quad Nil_V \qquad\quad \to Nil \\ &\quad Cons_V\ x\ xs \to reverse\ xs :+\!\!+ wrap\ x\end{aligned}$$

In general, doing pattern matching in terms of **case**s is not entirely satisfactory because **case**s do not nest as elegantly as left-hand-side patterns. Nevertheless, the GHC extensions `ViewPatterns` and `PatternSynonyms` add syntactic sugar that allows us to pattern-match on the left-hand side. Using these extensions we can make the

definition of $reverse$ look almost like the original. First, we declare a pattern synonym for each constructor:

$$\begin{aligned}&\mathbf{pattern}\ \underline{Nil} &&\leftarrow (view_L \rightarrow Nil_V)\\ &\mathbf{pattern}\ \underline{Cons}\ x\ xs &&\leftarrow (view_L \rightarrow Cons_V\ x\ xs)\end{aligned}$$

The pattern synonyms state that pattern matching on $\underline{Nil}$ is the same as applying $view_L$ and pattern matching the result on Nil_V, and that pattern matching on $\underline{Cons}\ x\ xs$ is the same as applying $view_L$ and pattern matching the result on $Cons_V\ x\ xs$.

After sugaring, the function $reverse$ is simply:

$$\begin{aligned}&reverse\ \underline{Nil} &&= Nil\\ &reverse\ (\underline{Cons}\ x\ xs) &&= reverse\ xs :{+\!\!+}\ wrap\ x\end{aligned}$$

The only missing piece, the definition of $view_L$, is straightforward.

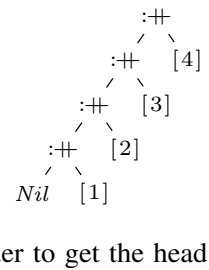
$$\begin{aligned}&view_L :: List\ a \rightarrow ListView\ a\\ &view_L\ Nil &&= Nil_V\\ &view_L\ (Cons\ x\ xs) &&= Cons_V\ x\ xs\\ &view_L\ (\underline{Nil} :{+\!\!+}\ ys) &&= view_L\ ys\\ &view_L\ (\underline{Cons}\ x\ xs :{+\!\!+}\ ys) &&= Cons_V\ x\ (xs :{+\!\!+}\ ys)\end{aligned}$$

As opposed to basic lists, the computation of concatenations happens when we inspect a list with $view_L$, rather than when we construct it. Note that the last two equations of $view_L$, which match on $:{+\!\!+}$, mimic the definition of $+\!\!+$ for basic lists.

In this implementation, concatenation is cheap. Unfortunately, views are expensive. After applying $reverse$ to the list

$$Cons\ 1\ (Cons\ 2\ (Cons\ 3\ (Cons\ 4\ Nil)))$$

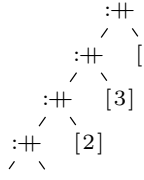
we get the following tree:

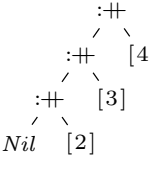


Applying $view_L$ in order to get the head and tail is linear in the length of the list, as it requires traversing the left-spine.

$$
\begin{aligned}
&\textbf{data}\ List\ a = Nil \\
&\qquad\qquad \mid\ Cons\ a\ (List\ a) \\
&\qquad\qquad \mid\ List\ a :{+\!\!+}\ List\ a \\
&view_L :: List\ a \to ListView\ a \\
&view_L\ Nil = Nil_V \\
&view_L\ (Cons\ x\ xs) = Cons_V\ x\ xs \\
&view_L\ ((xs :{+\!\!+}\ ys) :{+\!\!+}\ zs) = view_L\ (xs :{+\!\!+}\ (ys :{+\!\!+}\ zs)) \\
&view_L\ (\underline{Nil} :{+\!\!+}\ ys) = view_L\ ys \\
&view_L\ (\underline{Cons}\ x\ xs :{+\!\!+}\ ys) = Cons_V\ x\ (xs :{+\!\!+}\ ys)
\end{aligned}
$$

Figure 1. Definition of lists with a smart view

$view_L$ (


$= Cons_V\ 1$ (


)

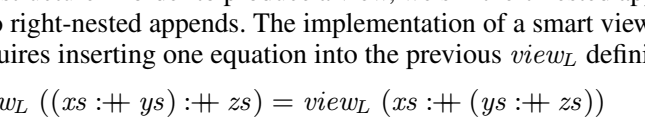
Therefore, the function $reverse \circ reverse$ is quadratic on the length of the input list, as the first $reverse$ necessarily yields left-nested appends, and then each $view_L$ in the second $reverse$ needs to traverse the whole list.

Our solution to this problem is to use a *smart view*: as we traverse the structure in order to produce a view, we shift left-nested appends into right-nested appends. The implementation of a smart view only requires inserting one equation into the previous $view_L$ definition:

$$view_L\ ((xs :{+\!\!+}\ ys) :{+\!\!+}\ zs) = view_L\ (xs :{+\!\!+}\ (ys :{+\!\!+}\ zs))$$

Figure 1 contains the complete definition of lists with a smart view.

Now $reverse \circ reverse$ is linear on the length of the list, as we only pay once for the traversal of left-nested appends. This is illustrated clearly by the following example: applying $view_L$ to a left-leaning list yields a right-leaning tail:



Note that *internally* $List$s are not lists but trees. Several equations that we expect to hold for lists do not hold for $List$s. For example, $(xs :{+\!\!+}\ ys) :{+\!\!+}\ zs$ is distinct from $xs :{+\!\!+}\ (ys :{+\!\!+}\ zs)$ as they are distinct trees. However, it is precisely the ability to make this distinction that

enables the optimisation provided by the extra equation in $view_L$. Moreover, for programmes that only inspect $List$s by using $view_L$, $List$s are indistinguishable from ordinary lists. Hence, $List$s are lists *observationally*.

A smart view modifies the structure when we inspect it. In that sense, smart views are reminiscent of splay trees, with whom they share many of their advantages and disadvantages. On the plus side, they are fast and simple. On the minus side, they are efficient only with respect to single-future amortised time: given a left-leaning list xs, we are going to pay the price of pattern-matching xs every time we execute $view_L\ xs$.

3. Efficient Free Monads

We generalise the idea of smart views on lists to other data structures where operations are more efficient when associated in a particular way. In this section we improve the slow bind operation of the free monad with a smart view that keeps the free monad ability to do monadic reflection efficiently.

3.1 Basic Free Monad

An important example of a data structure where the associativity of an operation determines its efficiency is that of a general leaf-labelled tree known as the *free monad*. Free monads are very useful for representing abstract syntax trees, where operations of the language are nodes in the tree and variables are labels in the leaves. The bind of this monad implements simultaneous substitution, which in this representation is given by *grafting* the trees, i.e. extending a tree by replacing each leaf by a tree.

The basic implementation of a free monad is the following:

$$\begin{aligned}&\textbf{data}\ \mathit{Free}\ f\ a = \mathit{Var}\ a\\ &\qquad\qquad\mid \mathit{Con}\ (f\ (\mathit{Free}\ f\ a))\end{aligned}$$

An element of $\mathit{Free}\ f\ a$ consists of a tree with f-nodes and leaves with a values. This datatype yields a monad for every functor f:

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Monad}\ (\mathit{Free}\ f)\ \textbf{where}\\ &\quad \mathit{return} \quad\ = \mathit{Var}\\ &\quad \mathit{Var}\ a \gg\!= f = f\ a\\ &\quad \mathit{Con}\ t \gg\!= f = \mathit{Con}\ (\mathit{fmap}\ (\gg\!= f)\ t)\end{aligned}$$

An important concern in a free monad implementation is the efficiency of the bind operation. As with list concatenation, the bind operation above is inefficient when left-nested, since it traverses the tree until it gets to the leaves, and as a consequence, the evaluation of the expression $(t \gg\!= f) \gg\!= g$ will traverse t twice, due to the left-nested binds.

3.2 A Smart View on Free Monads

The usual solution to the inefficiency of left-nested binds in the free monad is to apply the codensity transformation [5, 15], but this transformation does not allow for pattern-matching on the constructors of the free monad without losing the efficiency gains. In order to have both an efficient bind and an efficient view, we apply the same recipe as for lists:

- We add a constructor for the bind operation.
- We define a view function that shifts binds to the right.

Figure 2 provides the full definition of a free monad with a smart

view. The differences with respect to the basic implementation of the free monad are that a constructor $:\gg=$ is added and is used to implement the bind of the monad, and that pattern-matching is done using $view_F$. Once again, computation is performed when inspecting rather than when constructing the tree. The cases in the definition of $view_F$ for the $:\gg=$ pattern mimic the definition of the bind for the basic free monad, except for the additional equation shifting left-nested binds to the right.

As it happened with the smart-view implementation of lists, which was not a list internally, the type $Free\ f$ is not a monad internally. For instance, the associativity law of bind does not hold for the declared instance. However, distinguishing the two manners in which we can associate bind is precisely what we need in order to shift binds to the right and make views efficient. By restricting access to the internal representation with $view_F$, $Free\ f$ is a monad observationally.

4. Efficient Non-determinism Monads

Another structure where the associativity of an operation is important is non-determinism monads. These monads not only need an efficient bind, but also need an efficient choice operator. We analyse

$$\begin{aligned}&\textbf{data}\ Free\ f\ x = Var\ x\\ &\qquad\qquad \mid\ Con\ (f\ (Free\ f\ x))\\ &\qquad\qquad \mid\ \forall a.(Free\ f\ a) :\gg= (a \to Free\ f\ x)\end{aligned}$$

$$\begin{aligned}&\textbf{instance}\ Monad\ (Free\ f)\ \textbf{where}\\ &\quad return = Var\\ &\quad (\gg=)\ \ = (:\gg=)\end{aligned}$$

$$\textbf{data}\ FreeMonadView\ f\ a = Var_V\ a \mid Con_V\ (f\ (Free\ f\ a))$$

$$
\begin{array}{l}
\textbf{pattern}\ \underline{Var}\ a \leftarrow (view_F \rightarrow Var_V\ a) \\
\textbf{pattern}\ \underline{Con}\ t \leftarrow (view_F \rightarrow Con_V\ t) \\
view_F\ (Var\ a) \qquad\qquad\quad = Var_V\ a \\
view_F\ (Con\ t) \qquad\qquad\quad = Con_V\ t \\
view_F\ ((m :\gg\!= f) :\gg\!= g) = view_F\ (m :\gg\!= \lambda x \rightarrow f\ x :\gg\!= g) \\
view_F\ (\underline{Var}\ a :\gg\!= f) \quad = view_F\ (f\ a) \\
view_F\ (\underline{Con}\ t :\gg\!= f) \quad = Con_V\ (fmap\ (:\gg\!= f)\ t)
\end{array}
$$

Figure 2. Free monad with a smart view.

two different inefficient implementations and show how they can be improved using a smart view.

4.1 Basic List Monad Transformer

The list monad transformer [6] is often used to model the combination of non-determinism and other effects. For every monad m, the monad transformer yields a non-determinism monad $ListT\ m$. Its definition is as follows:

$$
\begin{array}{l}
\textbf{newtype}\ ListT\ m\ a = LT\ (m\ (Maybe\ (a, ListT\ m\ a))) \\
view_{LT} :: ListT\ m\ a \rightarrow m\ (Maybe\ (a, ListT\ m\ a)) \\
view_{LT}\ (LT\ x) = x
\end{array}
$$

Its *Monad* instance states that $ListT\ m$ is a monad for every monad m. The bind operation is defined in terms of the *mplus* operation, which is given below.

$$
\begin{array}{l}
\textbf{instance}\ Monad\ m \Rightarrow Monad\ (ListT\ m)\ \textbf{where} \\
\quad return\ x = LT\ (return\ (Just\ (x, mzero)))
\end{array}
$$

$$
\begin{aligned}
m \gg\!= f \;\; = \; & LT\ (view_{LT}\ m \gg\!= \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of} \\
& \quad Nothing \quad \rightarrow return\ Nothing \\
& \quad Just\ (h, t) \rightarrow view_{LT}\ (f\ h\ `mplus`\ (t \gg\!= f)))
\end{aligned}
$$

The $MonadTrans$ instance states that $ListT$ is a monad transformer and has a monad morphism

$$lift :: Monad\ m \Rightarrow m\ a \rightarrow ListT\ m\ a$$

lifting computations from the underlying monad into the transformed monad.

$$
\begin{aligned}
&\mathbf{instance}\ MonadTrans\ ListT\ \mathbf{where} \\
&\quad lift\ m = LT\ (m \gg\!= \lambda x \rightarrow return\ (Just\ (x, mzero)))
\end{aligned}
$$

The list monad transformer implements two operations for nondeterminism, which are specified by the $MonadPlus$ interface.

$$
\begin{aligned}
&\mathbf{class}\ Monad\ m \Rightarrow MonadPlus\ m\ \mathbf{where} \\
&\quad mzero :: m\ a \\
&\quad mplus :: m\ a \rightarrow m\ a \rightarrow m\ a
\end{aligned}
$$

The operation $mplus$ chooses between two computations, and $mzero$ represents the empty choice. The corresponding implementations in $ListT$ are as follows:

$$
\begin{aligned}
&\mathbf{instance}\ Monad\ m \Rightarrow MonadPlus\ (ListT\ m)\ \mathbf{where} \\
&\quad mzero \qquad\quad = LT\ (return\ Nothing) \\
&\quad m\ `mplus`\ n = LT\ (view_{LT}\ m \gg\!= \lambda x \rightarrow \mathbf{case}\ x\ \mathbf{of} \\
&\qquad\qquad\qquad Nothing \quad \rightarrow view_{LT}\ n \\
&\qquad\qquad\qquad Just\ (h, t) \rightarrow return\ (Just\ (h, t\ `mplus`\ n)))
\end{aligned}
$$

A monad which is an instance of $MonadPlus$ and additionally, implements the $MonadLogic$ interface, supports operators for fair

disjunction, fair conjunction, conditionals, and pruning [7].

$$\textbf{class}\ \mathit{MonadPlus}\ m \Rightarrow \mathit{MonadLogic}\ m\ \textbf{where}$$
$$\quad \mathit{msplit} :: m\ a \to m\ (\mathit{Maybe}\ (a, m\ a))$$

The $\mathit{MonadLogic}$ instance of ListT follows directly from the view_{LT} operation and the fact that ListT is a monad transformer (and therefore implements *lift*.)

$$\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadLogic}\ (\mathit{ListT}\ m)\ \textbf{where}$$
$$\quad \mathit{msplit}\ x = \mathit{lift}\ (\mathit{view}_{LT}\ x)$$

The main problem with the basic list monad transformer is that left-nested *mplus* operations are inefficient. We solve this problem with a smart view.

4.2 A Smart View on the List Monad Transformer

In order to obtain a smart view on the list monad transformer, we follow the same recipe as before. First, we add a constructor $(:+)$ corresponding to the *mplus* operation:

$$\begin{aligned}\textbf{data}\ \mathit{ListT}\ m\ a = {} & \mathit{LT}\ (m\ (\mathit{Maybe}\ (a, \mathit{ListT}\ m\ a))) \\ & \mid (\mathit{ListT}\ m\ a) :+ (\mathit{ListT}\ m\ a)\end{aligned}$$

Next, we change the view_{LT} function so that it performs the computation corresponding to *mplus* while shifting left-nested operations to the right.

$$\begin{aligned}&\mathit{view}_{LT} :: \mathit{Monad}\ m \Rightarrow \\ &\qquad \mathit{ListT}\ m\ a \to m\ (\mathit{Maybe}\ (a, \mathit{ListT}\ m\ a)) \\ &\mathit{view}_{LT}\ (\mathit{LT}\ v) && = v \\ &\mathit{view}_{LT}\ ((m :+ n) :+ o) && = \mathit{view}_{LT}\ (m :+ (n :+ o)) \\ &\mathit{view}_{LT}\ (m :+ n) && = \mathit{view}_{LT}\ m \gg\!= \lambda x \to \textbf{case}\ x\ \textbf{of}\end{aligned}$$

$$
\begin{array}{l}
\mathit{Nothing} \quad \rightarrow \mathit{view}_{LT}\ n \\
\mathit{Just}\ (h, t) \rightarrow \mathit{return}\ (\mathit{Just}\ (h, t\ `\mathit{mplus}`\ n))
\end{array}
$$

Last, we change the $\mathit{MonadPlus}$ instance so that it uses the newly added constructor.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadPlus}\ (\mathit{ListT}\ m)\ \textbf{where} \\
\quad \mathit{mzero} = \mathit{LT}\ (\mathit{return}\ \mathit{Nothing}) \\
\quad \mathit{mplus} = (:+)
\end{array}
$$

No more changes are needed! The Monad, $\mathit{MonadTrans}$, and $\mathit{MonadLogic}$ instances are exactly the same as before. With a few simple changes, we have obtained a list monad transformer with efficient mplus and reflection.

4.3 Free MonadPlus

The second instance of a non-determinism monad that we are going to analyse is that of the free $\mathit{MonadPlus}$. In Section 3, we showed how the free monad constructs a monad for every functor. Analogously, the free $\mathit{MonadPlus}$ construction yields a $\mathit{MonadPlus}$ for every functor.

The free $\mathit{MonadPlus}$ is given by the following datatype.

$$
\begin{array}{l}
\textbf{data}\ \mathit{FMP}\ f\ x = \mathit{FNil} \\
\qquad\qquad\qquad\quad \mid\ \mathit{ConsV}\ x\ (\mathit{FMP}\ f\ x) \\
\qquad\qquad\qquad\quad \mid\ \mathit{ConsF}\ (f\ (\mathit{FMP}\ f\ x))\ (\mathit{FMP}\ f\ x)
\end{array}
$$

Its Monad instance is the following:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Monad}\ (\mathit{FMP}\ f)\ \textbf{where} \\
\quad \mathit{return}\ x \qquad\qquad\quad = \mathit{ConsV}\ x\ \mathit{FNil} \\
\quad \mathit{FNil} \qquad\quad \gg\!= f = \mathit{FNil}
\end{array}
$$

$$(ConsV\ x\ v) \gg\!= f = f\ x\ `mplus`\ (v \gg\!= f)$$
$$(ConsF\ t\ v) \gg\!= f = ConsF\ (fmap\ (\gg\!= f)\ t)\ (v \gg\!= f)$$

Whereas in the free monad we have a choice of a Var or a Con, in the free $MonadPlus$ we have a list of those two choices. Elements of the list are added by $ConsV$ and $ConsF$, and $FNil$ signals the empty list. The bind operation is applied to each element of the list. The corresponding $MonadPlus$ instance is as follows:

$$\begin{aligned}
&\textbf{instance}\ MonadPlus\ (FMP\ f)\ \textbf{where}\\
&\quad mzero &&= FNil\\
&\quad FNil\ \ \ `mplus`\ y &&= y\\
&\quad (ConsV\ x\ y)\ `mplus`\ z &&= ConsV\ x\ (y\ `mplus`\ z)\\
&\quad (ConsF\ x\ y)\ `mplus`\ z &&= ConsF\ \ x\ (y\ `mplus`\ z)
\end{aligned}$$

The free monad plus suffers from two deficiencies: both left-nested binds and left-nested $mplus$ are inefficient. We solve both problems with a smart view.

4.4 A Smart View on Free MonadPlus

In the smart view on the free $MonadPlus$, we need to solve the associativity problem of two operations, and therefore we add two constructors:

$$\begin{aligned}
&\textbf{data}\ FMP\ f\ x = FNil\\
&\qquad\qquad\ \ \mid ConsV\ x\ (FMP\ f\ x)\\
&\qquad\qquad\ \ \mid ConsF\ (f\ (FMP\ f\ x))\ (FMP\ f\ x)\\
&\qquad\qquad\ \ \mid (FMP\ f\ x) :+ (FMP\ f\ x)\\
&\qquad\qquad\ \ \mid \forall a.(FMP\ f\ a) :\gg\!= (a \to FMP\ f\ x)
\end{aligned}$$

The $Monad$ and $MonadPlus$ instances now simply use the

newly added constructors:

$$
\begin{aligned}
&\textbf{instance } \mathit{Monad}\ (\mathit{FMP}\ f)\ \textbf{where}\\
&\quad \mathit{return}\ x = \mathit{ConsV}\ x\ \mathit{FNil}\\
&\quad (\gg\!=) \quad\ = (:\gg\!=)\\
&\textbf{instance } \mathit{MonadPlus}\ (\mathit{FMP}\ f)\ \textbf{where}\\
&\quad \mathit{mzero} \quad\ = \mathit{FNil}\\
&\quad x\ \text{`}\mathit{mplus}\text{`}\ y = x :+ y
\end{aligned}
$$

We define a view datatype for recovering reflection, along with pattern synonyms that add syntactic sugar.

$$
\begin{aligned}
&\textbf{data } \mathit{View}_M\ f\ x = \mathit{FNil}_V\\
&\qquad\qquad\quad \mid\ \mathit{ConsV}_V\ x\ (\mathit{FMP}\ f\ x)\\
&\qquad\qquad\quad \mid\ \mathit{ConsF}_V\ (f\ (\mathit{FMP}\ f\ x))\ (\mathit{FMP}\ f\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{pattern } \underline{\mathit{FNil}} \qquad\quad \leftarrow (\mathit{view}_M \rightarrow \mathit{FNil}_V)\\
&\textbf{pattern } \underline{\mathit{ConsV}}\ x\ xs \leftarrow (\mathit{view}_M \rightarrow \mathit{ConsV}_V\ x\ xs)\\
&\textbf{pattern } \underline{\mathit{ConsF}}\ t\ xs\ \leftarrow (\mathit{view}_M \rightarrow \mathit{ConsF}_V\ t\ xs)
\end{aligned}
$$

As before, the view function turns left-associated operations into right-associated operations. In this case it needs to do it for both left-associated occurrences of :+ and left-associated occurrences of :$\gg\!=$. When the operations are not left-associated, then the view performs the computations that were done in the original definitions of *mplus* and bind.

$$
\begin{aligned}
&\mathit{view}_M :: \mathit{Monad}\ f \Rightarrow \mathit{FMP}\ f\ x \rightarrow \mathit{View}_M\ f\ x\\
&\mathit{view}_M\ \mathit{FNil} \qquad\quad = \mathit{FNil}_V\\
&\mathit{view}_M\ (\mathit{ConsV}\ x\ xs) \quad = \mathit{ConsV}_V\ x\ xs\\
&\mathit{view}_M\ (\mathit{ConsF}\ x\ xs) \quad = \mathit{ConsF}_V\ x\ xs
\end{aligned}
$$

$$
\begin{aligned}
view_M\ ((x :+ y) :+ z) &= view_M\ (x :+ (y :+ z)) \\
view_M\ (\underline{FNil} :+ y) &= view_M\ y \\
view_M\ (\underline{ConsV}\ x\ xs :+ y) &= ConsV_V\ x\ (xs :+ y) \\
view_M\ (\underline{ConsF}\ x\ xs :+ y) &= ConsF_V\ x\ (xs :+ y) \\
view_M\ ((m :\gg\!= f) :\gg\!= g) &= \\
&\qquad view_M\ (m :\gg\!= (\lambda x \to f\ x :\gg\!= g)) \\
view_M\ (\underline{FNil} :\gg\!= f) &= FNil_V \\
view_M\ (\underline{ConsV}\ x\ xs :\gg\!= f) &= view_M\ (f\ x :+ (xs :\gg\!= f)) \\
view_M\ (\underline{ConsF}\ t\ xs :\gg\!= f) &= ConsF_V\ (fmap\ (:\gg\!= f)\ t) \\
&\qquad (xs :\gg\!= f)
\end{aligned}
$$

As this last example shows, the same procedure for optimising one operation can be applied when we want to optimise two or more operations.

5. Related Work

The search for lists with fast concatenation is a well-known problem for which many solutions have been proposed in the past. Additionally, some of the work has also been generalised to monads, and at least in one case to $MonadPlus$. We discuss some of the most relevant related works.

5.1 Modified Reduction Semantics

Sleep and Holmström [11] solve the problem of left-nested appends by means of an interpreter for a lazy evaluator which regards the $+\!\!+$ operator as a constructor with a special reduction semantics. This reduction semantics shifts left-nested appends into right-nested appends, achieving the same effect as the smart view of Section 2.2.

This approach requires a lazy language, in contrast to smart

views which also work in a strict setting. The difference is that in this approach the use of the associativity of append is commanded by the evaluation of the list, whereas in the smart view approach it is commanded by invocation of a *view* function.

There are other approaches that, like [11], solve the problem of left-nested appends by modifying the semantics [14, 16]. In these approaches, the whole program needs to be transformed or compiled in a special way.

In the related work that follows a different approach is taken. The starting point is an abstract data type, and therefore only the abstract data type implementation needs to be changed.

5.2 Catenable Lists

The search for catenable lists has a long history. The most relevant work for Haskell implementations are the catenable double-ended queues in Okasaki's book [9] and finger trees [3], which is the data structure chosen in Haskell's *Data.Sequence* package. Both of these structures do more than just fast concatenation and views, as they implement double-ended queues.

However, if one can do without the extra functionality and single future amortised time is enough, lists with a smart view cannot be beat for simplicity and speed (see Section 6).

The simplicity of structures with a smart view is an important factor when one wants to reproduce the optimisation in data structures other than lists.

5.3 Continuation-passing Representations

Cayley lists (also known as difference lists or Hughes' lists [4]) are a good way to speed up concatenations. It is a simple approach

which has been also applied to the optimisation of the bind in the free monad through the codensity monad transformation [5, 15]. Moreover, it has been shown that the approach is an instance of a generic Cayley representation for monoids, which means that it can be applied to other structures such as applicative functors [10]. Continuation-based implementations have also been proposed for non-determinism monad transformers [2, 7].

The main limitation of all these continuation-passing representations is that they lack support for pattern-matching. This means that they will work well if all the computation can be performed without inspecting the structure, and only in the end the results are analysed. The benefits are lost if one needs to inspect the structure in the middle of the computation.

5.4 Explicit Binds

Uustalu introduced an approach of "explicit binds" which is quite close to ours [12]. In the explicit-bind approach, the operation that one wants to optimise is introduced as a constructor in exactly the same way that one does in the smart-view approach. However, as opposed to the smart-view approach, the data structure is inspected using a special fold operator that applies the selected operation in the most efficient order. For example, for lists the fold operation would be:

$$
\begin{aligned}
&\mathit{foldE} :: (a \to b \to b) \to b \to \mathit{List}\ a \to b \\
&\mathit{foldE}\ h\ e\ \mathit{Nil} &&= e \\
&\mathit{foldE}\ h\ e\ (\mathit{Cons}\ x\ xs) &&= h\ x\ (\mathit{foldE}\ h\ e\ xs) \\
&\mathit{foldE}\ h\ e\ (xs :{+\!\!+}\ ys) &&= \mathit{foldE}\ h\ (\mathit{foldE}\ h\ e\ ys)\ xs
\end{aligned}
$$

The disadvantage of this approach is that one is required to

write functions in terms of folds, instead of using pattern-matching. Uustalu also defines a primitive-recursion operator:

$$
\begin{aligned}
&primrec :: (a \to b \to List\ a \to b) \to b \to List\ a \to b \\
&primrec\ h\ e\ Nil &&= e \\
&primrec\ h\ e\ (Cons\ x\ xs) &&= h\ x\ (primrec\ h\ e\ xs)\ xs \\
&primrec\ h\ e\ (xs :{+\!\!+} ys) &&= primrec\ h'\ (primrec\ h\ e\ ys)\ xs \\
& && \mathbf{where}\ h'\ x\ a\ xs = h\ x\ a\ (xs :{+\!\!+} ys)
\end{aligned}
$$

which in principle would allow us to define a view:

$$view_L = primrec\ (\lambda x\ \_\ xs \to Cons_V\ x\ xs)\ Nil_V$$

However, this $view_L$ is equivalent to our first, unoptimised implementation. That is, if we define $reverse$ by pattern-matching on this view, $reverse \circ reverse$ is quadratic.

The solution to this problem is to use a smart view in the definition of $primrec$: we add another equation turning left-nested appends into right-nested appends. Joining the two approaches yields the following definition:

$$
\begin{aligned}
&primrec'\ h\ e\ Nil &&= e \\
&primrec'\ h\ e\ (Cons\ x\ xs) &&= h\ x\ (primrec'\ h\ e\ xs)\ xs \\
&primrec'\ h\ e\ ((xs :{+\!\!+} ys) :{+\!\!+} zs) && \\
& && = primrec'\ h\ e\ (xs :{+\!\!+} (ys :{+\!\!+} zs)) \\
&primrec'\ h\ e\ (xs :{+\!\!+} ys) &&= primrec'\ h'\ (primrec'\ h\ e\ ys)\ xs \\
&\quad \mathbf{where}\ h'\ x\ a\ xs = h\ x\ a\ (xs :{+\!\!+} ys) &&
\end{aligned}
$$

Therefore, one can use both approaches simultaneously. After all, giving $foldE$ and $primrec'$ access to the internal representation cannot hurt. Perhaps surprisingly, the addition of these operations is inconsequential. Our benchmarks show that functions implemented

using $foldE$ perform as well as functions that use the following $foldr$ defined in terms of pattern-matching on $view_L$.

$$
\begin{aligned}
&foldr :: (a \to b \to b) \to b \to List\ a \to b \\
&foldr\ h\ e\ \underline{Nil} &&= e \\
&foldr\ h\ e\ (\underline{Cons}\ x\ xs) &&= h\ (foldr\ h\ e\ xs)
\end{aligned}
$$

5.5 Monadic Reflection

Van der Ploeg and Kiselyov [13] propose a data structure that solves exactly the problem that we address with smart views: optimising an operation such as bind in the free monad, or $mplus$ for non-determinism monads, without losing the ability to pattern-match efficiently. The technique generalises an efficient data structure for lists to monads by keeping a type aligned sequence of monadic binds. Because of the type aligned sequences, the implementation is much more complex than the smart view implementation. Moreover, benchmarks show that smart views are noticeably faster.

5.6 Operational Monad

The smart view on the free monad shown in Section 3 is very similar to the implementation of the free monad in the `operational` package. Note that this is quite a different implementation from the one described by its author in a tutorial article [1]. The package also provides an implementation of a non-determinism monad, but this implementation does not use smart views and suffers from quadratic time on left-nested applications of $mplus$.

6. Benchmarks

We provide some micro-benchmarks in order to give an idea of what performance can be expected from the use of smart views.

All benchmarks where done on an Intel Core i5-3330 CPU, with 16GB of RAM. All programs were compiled using GHC 7.8.2 with optimisation turned on. For obtaining the running times we used the `criterion` package, which executes each test several times in order to account for accidental differences in CPU load.

In each benchmark, we compare implementations with a linear asymptotic complexity, leaving out implementations which are quadratic for that test. We express the results as relative time with respect to the fastest implementation.

The source code for the benchmarks can be downloaded from `http://www.fceia.unr.edu.ar/~mauro/pubs/smartviews`.

6.1 Lists

We compare lists with a smart view (Section 2.2) against Okasaki's catenable double-ended queues [9] and Finger Trees [3]. We benchmarked the running time of the function $reverse \circ reverse$ which mixes pattern-matching and concatenation, for input of different lengths. The implementation using smart views is the fastest, with catenable double-ended queues being 4.6 times slower, and finger trees being 1.5 times slower.

Both catenable deques and finger trees implement efficiently the removal of the last element, an operation which is inefficient for lists with a smart view. However, if pattern-matching and concatenation is all that is needed, the smart-view implementation seems to be the fastest.

We also compare two implementations of fold for lists with a smart view. One is the fold with access to the internal representation,

as presented by Uustalu [12] (see Section 5.4), and the other is *foldr* written using views.

The benchmark compares the performance of summing a list of integers by writing *sum* as a fold. Both implementations show very close running times (difference less than 2%), and therefore we conclude that we would not gain much by adding a fold with access to the internal representation.

6.2 Free Monads

We compare the following implementations of free monads:

- Free monad with a smart view (Section 3.2).
- **Codensity:** Codensity monad on a free monad [5, 15].
- **Ref:** Free monad using the "reflection without remorse" technique [13].
- **Oper:** Free monad from the `operational` package.

While all of these implementations deal efficiently with left-nested binds, the codensity monad is the only one that does not have an efficient reflection mechanism.

We measure the running time of the function *fullTree* [15]. This is a toy example which constructs a binary tree using left-nested binds, which then is consumed with a zig-zag traversal. This benchmark does not use reflection, so we expected the codensity transformation to be the fastest. However, even in this case, the smart-view implementation is the fastest, , with **Codensity** being 1.2 times slower, **Oper** being 1.5 times slower, and **Ref** being 2.9 times slower.

Next, we measure the running time of the function *interleave*,

which interleaves two monadic computations making heavy use of reflection (and therefore we left out the codensity transformation). Again, the free monad with a smart view is the fastest, with **Oper** being 1.2 time slower, and **Ref** being 2.2 times slower.

6.3 Non-determinism Monads

Last, we test implementations of non-determinism monads with three benchmarks. The implementations we compare are:

- List monad transformer with a smart view (Section 4.2).
- Free $MonadPlus$ with a smart view (Section 4.4).
- **Ref:** List monad transformer using "reflection without remorse" [13].
- **LogicT:** Backtracking monad transformer based on continuations [7]. This implementation deals with left-nested $mplus$ efficiently, but poorly with reflection.
- **ListT:** Basic list monad transformer.

In the first benchmark, we compare the running times of different implementations when observing all results in left-nested applications of $mplus$. The unoptimised list transformer is not included since it takes quadratic time. Surprisingly, the two implementations that use smart views even best the continuation-based implementation. More concretely, in this test the fastest implementation is the Free MonadPlus with a smart view, followed by the list monad transformer with a smart view (1.2 times slower), then **LogicT** (1.4 times slower), and finally **Ref** (4.8 times slower). Note that this is just a micro-benchmark. We still expect the continuation-based implementation to be faster in real applications where reflection is

not needed.

In the second benchmark, we evaluate taking the first n results from a computation. This test does use reflection, and therefore **LogicT** takes quadratic time rather than linear, so it is not included in the comparison. We do include the original list transformer **ListT**, which, as expected, performs well in this test. The smart view free monad plus is the fastest, followed by the basic list transformer (1.5 times slower), then the smart view list transformer (2 times slower), and finally **Ref** (4.2 times slower).

In the third benchmark, we test the fair conjunction operation, which uses reflection. Again, smart views have the advantage. The smart view free $MonadPlus$ is the fastest, with the smart view list transformer being 1.5 times slower, and **Ref** being 2.7 times slower. Compared with the "reflection without remorse" technique, smart views obtain similar asymptotic complexity but, perhaps due to their simplicity, much lower constants.

6.4 Smart Views in Strict Languages

The smart view technique also works in a strict setting. In order to validate this claim, we have implemented the smart view for lists in the strict functional language ML. As it was done in Section 6.1, we tested the implementation with the function $reverse \circ reverse$. As expected, the benchmarks show that the function runs in linear time. Moreover, when compared with an implementation of Okasaki's catenable deques the constant speedups are similar (catenable deques are 4 times slower in this test). Also, we obtained results similar to the Haskell case when running the benchmark that compares two implementations of $fold$, with and without access to the internal representation. Benchmarks were compiled using Moscow ML compiler version 2.10.

7. Conclusion

We have shown a technique for optimising operations in a data structure, while keeping efficient pattern-matching. We have shown the technique by constructing efficient versions of catenable lists, reflective free monads, and two implementations of reflective non-determinism monads. The extension of the technique to other data structures seems trivial.

In all of our examples we have optimised an operation by using its associativity. However, the technique can be readily applied to other algebraic properties. For example, in the free $MonadPlus$ example, it is trivial to add an equation that distributes bind over $mplus$:

$$view_M\ ((x :+ y) :\gg\!= f) = view_M\ ((x :\gg\!= f) :+ (y :\gg\!= f))$$

Smart views are an efficient solution with respect to single-future amortised time, whose simplicity cannot be understated. In order to optimise a datatype using its algebraic properties, it is a good idea to have a smart view on it.

Acknowledgements

We would like to thank the anonymous referees for their helpful feedback. This work was partially funded by Agencia Nacional de Promoción Científica y Tecnológica (ANPCyT) PICT 2009-15.

References

[1] H. Apfelmus. The Operational Monad Tutorial. *The Monad.Reader*,

Issue 15, January 2010.

[2] R. Hinze. Deriving backtracking monad transformers. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 186–197, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6.

[3] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.

[4] J. Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[5] G. Hutton, M. Jaskelioff, and A. Gill. Factorising folds for faster functions. *Journal of Functional Programming*, 20(Special Issue 3-4): 353–373, 2010.

[6] M. Jaskelioff and E. Moggi. Monad transformers as monoid transformers. *Theoretical Computer Science*, 411(51-52):4441 – 4466, 2010.

[7] O. Kiselyov, C. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In O. Danvy and B. C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 192–203. ACM, 2005. ISBN 1-59593-064-7.

[8] C. Okasaki. Views for Standard ML. In *SIGPLAN Workshop on ML*, pages 14–23, 1998.

[9] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, New York, NY, USA, 1998. ISBN 0-521-63124-6.

[10] E. Rivas and M. Jaskelioff. Notions of computation as monoids. *CoRR*, abs/1406.4823, 2014. URL `http://arxiv.org/abs/1406.4823`. Submitted to the Journal of Functional Programming.

[11] M. R. Sleep and S. Holmström. A short note concerning lazy reduction

rules for append. *Software: Practice and Experience*, 12(11):1082–1084, 1982. ISSN 1097-024X.

[12] T. Uustalu. Explicit binds: Effortless efficiency with and without trees. In T. Schrijvers and P. Thiemann, editors, *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25*, volume 7294 of *Lecture Notes in Computer Science*, pages 317–331. Springer, 2012. ISBN 978-3-642-29821-9.

[13] A. van der Ploeg and O. Kiselyov. Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In W. Swierstra, editor, *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 133–144. ACM, 2014. ISBN 978-1-4503-3041-1.

[14] J. Voigtländer. Concatenate, reverse and map vanish for free. *SIGPLAN Not.*, 37(9):14–25, Sept. 2002. ISSN 0362-1340.

[15] J. Voigtländer. Asymptotic improvement of computations over free monads. In *Proceedings of the 9th International Conference on Mathematics of Program Construction*, MPC '08, pages 388–403, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70593-2.

[16] P. Wadler. The concatenate vanishes. Technical report, Department of Computer Science, Glasgow University, December 1987.