Reflection without Remorse

Revealing a hidden sequence to speed up monadic reflection

Atze van der Ploeg
Centrum Wiskunde & Informatica
ploeg@cwi.nl

Oleg Kiselyov
University of Tsukuba
oleg@okmij.org

Abstract

A series of list appends or monadic binds for many monads performs algorithmically worse when left-associated. Continuation-passing style (CPS) is well-known to cure this severe dependence of performance on the association pattern. The advantage of CPS dwindles or disappears if we have to examine or modify the intermediate result of a series of appends or binds, before continuing the series. Such examination is frequently needed, for example, to control search in non-determinism monads.

We present an alternative approach that is just as general as CPS but more robust: it makes series of binds and other such operations efficient regardless of the association pattern – and also provides efficient access to intermediate results. The key is to represent such a conceptual sequence as an efficient sequence data structure. Efficient sequence data structures from the literature are homogeneous and cannot be applied as they are in a type-safe way to series of monadic binds. We generalize them to *type aligned sequences* and show how to construct their (assuredly order-preserving) implementations. We demonstrate that our solution solves previously undocumented, severe performance problems in iteratees, LogicT transformers, free monads and extensible effects.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications – Applicative (functional) languages

Keywords performance; monads; reflection; data structures

1. Introduction

It is well-known that list-concatenation $(+\!\!+)$ is not efficient when its left argument is itself the result of a concatenation. A popular solution to this problem is to use continuation-passing style in the form of difference lists. We recall the problems of list-concatenation and how continuation-passing style remedies it in Sections 2 and 3 respectively. However, continuation-passing style only solves the performance problem for certain usage patterns: if we need to observe intermediate results of concatenations, or build concatenations with sub-lists of other concatenations, then performance quickly degenerates. In other words: continuation-passing style again leads to performance problems if we alternate between building and observing.

In this paper, we show that this pattern also occurs in many other situations, which at first blush have nothing to do with lists. In many implementations of monads (e.g., iteratees and non-determinism monads), a series of binds $(\gg\!=)$ or choices (mplus), is quite like a series of list appends: they perform badly when left-associated. Like with lists, continuation-passing style makes such series perform algorithmically well regardless of the association pattern [24]. However, several monads also support *monadic reflection* [6], a way to observe and modify (a representation of) the current state of the computation. For example, the current state of a non-deterministic computation may be observed as a stream of results. We may remove the top result and continue with the rest – which is exactly what is needed to implement committed choice [16]. Such monadic reflection destroys the performance advantage of the continuation-passing style. This paper shows that one does not have to regret reflection.

For lists, the solution to the append-and-observe problem is to use a more suited sequence data structure, i.e. one that supports both head/tail and append operations efficiently. Such data structures can give an asymptotic improvement over both regular lists and difference lists. The surprise of this paper is that such efficient data structures can also give an asymptotic improvement for other problematic occurrences of the build-and-observe pattern, in particular, monads and monadic reflection. The key insight is that we can reveal the hidden, abstract sequence of monadic binds: we can represent is as a concrete sequence. By then choosing the most suited sequence data structure for the problem at hand, performance can be greatly improved.

However, the literature on efficient sequences deals with homogeneous collections. In a 'sequence' of binds, the types of the 'elements' may vary. To solve this problem, we introduce a generalization of sequences called *type aligned sequences*: heterogeneous sequences where the types enforce the element order. In this way, we can solve the performance problem in any situation exhibiting the problematic pattern, in a completely type-safe way.

We were confronted with the performance problems of monadic reflection in projects using monadic functional reactive programming [23] and the parallel composition of iteratees [15]. These practical problems have motivated the present research. We have distilled the issue into a performance problem with simple tree substitutions, which helped us see how changing the data representation to use efficient sequences can improve performance. This not only solves the original problem, but also gives a drop-in replacement for free monads [22] with better performance characteristics than previous approaches: examining a free monad value and binding it are both efficient, letting us alternate between these operations without performance penalty. This improved free monad leads, among other things, to an implementation of extensible ef-

Haskell '14, September 6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-3041-1/14/09…$15.00.
http://dx.doi.org/10.1145/2633357.2633360

fects [17] in which a wider range of effects can be modeled efficiently.

We begin with some background: Section 2 recalls the problematic build-and-observe pattern in several guises, and we discuss continuation passing style and its performance problems in Section 3. Then we present our contributions:

- We present a solution to the build-and-observe problem for any monoid where left-associated expressions are more costly than right-associated expressions, giving an asymptotic running time improvement over both direct and continuation-passing style. (Section 4)
- We generalize our solution for monoids to monads, making left-associated bind expressions as well as monadic reflection efficient. (Section 4)
- We introduce type aligned sequences. As an example, we show an implementation of efficient type aligned queues. (Section 5)
- We show how our method solves previously undocumented, severe performance problems with monadic reflection in iteratees, LogicT transformers, free monads and extensible effects. (Section 6)

And in Section 7 we conclude.

The code accompanying this paper is available at:
`https://github.com/atzeus/reflectionwithoutremorse`
The code in this paper is in Haskell, but our approach can be used in any language with GADTs (indexed data types).

2. The problematic pattern and its cost

In this background section we recall the performance problems of associative operators that traverse their left argument but not their right argument. In particular, we discuss list concatenation, tree substitution and generic tree substitution. We recall that the running time cost of equivalent expressions involving such operators can differ asymptotically.

2.1 A first example: list concatenation

To analyze the performance problems of list concatenation, we recall the relevant standard definitions:

```
data [a] = [] | a : [a]

[]      ++ r = r
(h : t) ++ r = h : t ++ r
```

To append two lists, we must traverse all elements of the first list to arrive at the empty constructor at the end. Hence, reducing $x +\!\!+ y$ to normal form requires $|x| + 1$ case distinctions, from now on called steps, where $|x|$ is the length of x.

One might argue that this is not a problem: thanks to laziness, observing the head of $x +\!\!+ y$ is just observing the head of x, plus one extra step. To observe the *n*-th element of a list we must traverse the list anyway: concatenation just adds one extra step per element.

The real problem arises if the left argument is itself the result of a concatenation. For example, in the expression $(x +\!\!+ y) +\!\!+ z$, the list x must be traversed *twice*: it occurs twice in a left hand side argument to $+\!\!+$. Hence, this expression runs in in $2|x| + |y| + 2$ steps, whereas the *equivalent* expression $x +\!\!+ (y +\!\!+ z)$ runs in just $|x| + |y| + 2$ steps. In this way, a wrong grouping of expressions involving $+\!\!+$ can easily lead to severe performance problems, as we shall see in full generality in §2.4.

2.2 Another example: Tree substitution

A different guise of the same problem occurs with trees and an operation which substitutes the leaves of a tree with another tree:

```
data Tree = Node Tree Tree
          | Leaf

(↩) :: Tree → Tree → Tree
Leaf       ↩ y = y
(Node l r) ↩ y = Node (l ↩ y) (r ↩ y)
```

The performance situation is similar: evaluating $(x \hookleftarrow y) \hookleftarrow z$ traverses x twice, whereas the equivalent $x \hookleftarrow (y \hookleftarrow z)$ only traverses x once. Hence evaluating the former expression costs $|x|$ steps more than evaluating the latter, where $|x|$ is now the number of inner nodes in x.

For lists, this problem can be solved by simply using a catenable (meaning with fast concatenation) sequence data structure instead of a regular head-tail list. For trees, the solution is not so obvious. Should we investigate a new specialized data structure for trees or browse the literature to see if someone else has already invented it? (Hint: No.)

2.3 A Monadic example: Generic trees

The performance degradation from a bad association occurs not only with monoids, such as lists and trees. If we generalize our tree to a generic tree, with data at the leaves, then substitution becomes the monadic bind $(\gg\!=)$[1]:

```
data Tree a = Node (Tree a) (Tree a)
            | Leaf a

(↩) :: Tree a → (a → Tree b) → Tree b
(Leaf x)   ↩ f = f x
(Node l r) ↩ f = Node (l ↩ f) (r ↩ f)

instance Monad Tree where
  return = Leaf
  (>>=)  = (↩)
```

The performance situation is obviously the same: the only thing that changed is that $\hookleftarrow$ now takes a function as its right argument. Although $\hookleftarrow$ and $\gg\!=$ are not associative operators in the strict sense, they satisfy the similar associativity monad law:

$$(m \gg\!= f) \gg\!= g \quad \equiv \quad m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)$$

We now see that the situation is the same: $(m \gg\!= f) \gg\!= g$ runs in $|m|$ steps more than the equivalent $m \gg\!= (\lambda\, x \rightarrow f\, x \gg\!= g)$.

Note that while bind is not strictly an associative operator, the following operator, known as Kleisli composition, is strictly an associative operator:

```
(>>>) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c)
f >>> g = λx → f x >>= g
```

The similarity with the situation with lists and non-generic trees can then be made even stronger: $(p \ggg q) \ggg r$ is more costly than the equivalent $p \ggg (q \ggg r)$.

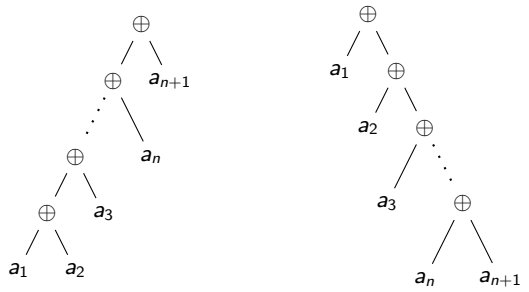
2.4 Asymptotic running time overhead

In general, the problem occurs with any *associative* (or satisfying the associativity monad law) operator $(\oplus)$ that traverses its left argument but not its right argument that operates on some *recursive*[2] data type. In this situation, $(x \oplus y) \oplus z$ costs $|x|$ more steps to evaluate than $x \oplus (y \oplus z)$, where $|x|$ is now the number of values of type X inside x that are non-terminal (i.e. they are not for example the empty list or a leaf).

Repeated application of such an operator can lead to asymptotic running time overhead if $|a \oplus b| \geq |a| + |b|$. For lists, this obviously

[1] This example is taken from [24].

[2] If the data type is not recursive, e.g., the Maybe monad, one can easily see that both left and right associations have the same asymptotic cost.



(a) A left-associated expression (b) A right-associated expression

Figure 1: Equivalent left- and right-associated expressions.

holds since $|\mathsf{a} +\!\!+ \mathsf{b}| = |\mathsf{a}| + |\mathsf{b}|$. For trees, the size of $\mathsf{a} \hookleftarrow \mathsf{b}$ is $|\mathsf{a}| + \mathsf{a}_l|\mathsf{b}|$, where a_l is the number of leaves in the tree a. Since there is at least one leaf in a tree, the inequality $|\mathsf{a} \hookleftarrow \mathsf{b}| \geq |\mathsf{a}| + |\mathsf{b}|$ holds.

That this leads to asympotic running time overhead can be seen as follows: a *left-associated expression*, as visualized in Figure 1(a):

$$(((a_1 \oplus a_2) \oplus a_3) \cdots \oplus a_n) \oplus a_{n+1}$$

then costs at least $\sum_{i=1}^{n-1}(n-i)|a_i|$ more steps than the equivalent *right-associated expression*, visualized in Figure 1(b):

$$a_1 \oplus (a_2 \oplus (a_3 \oplus \ldots (a_n \oplus a_{n+1}) \ldots))$$

If we assume that all elements have size one, i.e. $|a_i| = 1$, then we more easily see that a left-associated expression costs $O(n^2)$ more steps than a right-associated expression:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

Of course, these are the most extreme cases: most expressions will not be completely right- or left-associated. However, any expression that is not completely right-associated will yield an overhead. We cannot expect the programmer to only form right-associated expressions, especially when using laziness: the programmer must then make sure that every time the operator is used, the left hand side *cannot* be itself a result of this operator.

3. A popular partial solution: Continuation-passing style

In this second background section, we discuss a popular way to alleviate such performance problems for certain usage patterns, namely *continuation-passing style*. We illustrate this technique with difference lists, which use continuation-passing style to speed up list concatenation. We then show that difference lists only avoid performance problems if we do not alternate between building and observing and that the same holds for continuation-passing style in general.

3.1 Difference lists

The trick of difference lists [10] is to *only* build right-associated expressions. More precisely, difference lists are *functions* for building right-associated expressions, i.e. functions of the form:

$$\lambda\, t \;\rightarrow a_1 +\!\!+ (a_2 +\!\!+ (a_3 +\!\!+ (a_4 +\!\!+ \ldots +\!\!+ t)))$$

And hence we define difference lists as functions from lists to lists:

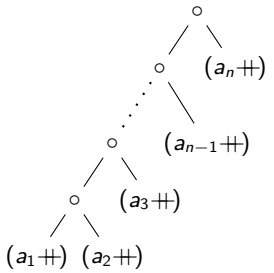


Figure 2: Difference list with worst case conversion characteristics.

```
type DiffList a = [a] → [a]
```

We can convert a difference list to a regular list by simply feeding it the empty list:

```
abs :: DiffList a → [a]
abs a = a []
```

To convert a list to a difference list, we partially apply $+\!\!+$:

```
rep :: [a] → DiffList a
rep = (++)
```

Concatenation is then simply function composition, since $(\mathsf{a} +\!\!+) \circ (\mathsf{b} +\!\!+) \equiv \lambda \mathsf{t} \rightarrow \mathsf{a} +\!\!+ (\mathsf{b} +\!\!+ \mathsf{t})$[3] :

```
(+̂+) :: DiffList a → DiffList a → DiffList a
(+̂+) = (∘)
```

The trick is then to concatenate using difference lists, and then convert the result to a list when needed. Since this will always produce a right-associated expression, the overhead associated with expressions that are not right-associated is avoided.

However, the problem with this technique is that converting a list to a difference list is expensive in the long run. Conversion of a list l to a difference list is simply $(\mathsf{l} +\!\!+)$, which, when the final result is observed, contributes the costs of $|\mathsf{l}|$ steps, adding one operation to each node in the list. Hence, if we convert back and forth n times, this will cost $n|\mathsf{l}|$ steps. Of course, converting the same list back and forth a number of times is a bit of a contrived situation. However, the problem also occurs if we convert a difference list to a list and convert *part* of the list back to a difference list.

Another, more subtle problem is that conversion in the other direction, from a difference list to a list, is not a constant time operation. We cannot *observe* anything directly on a difference list, for example we cannot see whether it is empty, and hence conversion to a regular list is often required. This conversion is not cheap: in the worst case the difference list consists of a left-associated expression of the following form, which is visualized in Figure 2:

$$((((a_1 +\!\!+) \circ (a_2 +\!\!+)) \circ (a_3 +\!\!+)) \ldots +\!\!+ (a_{n-1} +\!\!+)) \circ (a_n +\!\!+)$$

Converting such a difference list to list, by applying [] to it, then requires n invocations of $\circ$ to reduce to the following list expression:

$$a_0 +\!\!+ (a_1 +\!\!+ (a_2 +\!\!+ (a_3 +\!\!+ \ldots +\!\!+ (a_n +\!\!+ []))))$$

Only after these operations we can reduce further and inspect the resulting list to see whether it is empty or not. Hence, observing (parts of) intermediate lists can also lead to performance problems.

[3] We use the notation $(\mathsf{x} +\!\!+)$ as a shorthand for $(\lambda \mathsf{y} \rightarrow \mathsf{x} +\!\!+ \mathsf{y})$.

To summarize: difference lists only solve performance problems if our usage of lists is strictly separated into a build (i.e. concatenation) phase and an observation phase. If we alternate between building and observing, as is often needed, then performance problems will resurface.

3.2 General Continuation-passing style

The trick of difference lists, i.e. continuation-passing style, can be applied in many situations. For example, it can be applied to any monoid[4]:

```
type DiffMonoid a = a → a
abs :: Monoid a ⇒ DiffMonoid a → a
abs a = a mzero
rep :: Monoid a ⇒ a → DiffMonoid a
rep = mappend
instance Monoid a ⇒ Monoid (DiffMonoid a) where
  mempty  = id
  mappend = (∘)
```

If we apply the trick to monads, we get the codensity monad transformer [11], which is highly related to the continuation monad [18]:

```
type CodensityT m a = ∀ b. (a → m b) → m b
abs :: Monad m ⇒ CodensityT m a → m a
abs a = a return
rep :: Monad m ⇒ m a → CodensityT m a
rep = (≫=)
instance Monad m ⇒ Monad (CodensityT m) where
  return a = rep (return a)
    -- or equivalently: λk → k a
  m ≫= f = m ∘ flip f
    -- or equivalently: λk → m (λa → f a k)
```

The codensity monad transformer is often used for solving the performance problems of left-associated expressions [4, 24]. As with difference lists, this works fine if our usage is separated in a build and an observations phase. However, if we have another usage pattern, alternating between building and observing, the same problems as with difference lists occurs: continuation-passing style reintroduces performance problems.

4. Solving the problem

The main insight for our solution is that expressions of the form:

$$a_0 \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

are sequences and that such abstract sequences should be represented *explicitly*. With the previous approaches such sequences are only represented *implicitly*. More precisely, when directly using $\oplus$, these sequences are implicitly represented at runtime as trees where the leaves are the elements and nodes are (delayed) function applications. When using continuation-passing style, such sequences are also represented as trees, but now the leaves are functions representing the elements and the nodes are function composition. By making representation of these sequences explicit, we can choose a more suited sequence data structure and performance problems can be solved for any usage pattern.

We first illustrate our solution by applying it to tree substitution. We then show that applying our solution to generic trees requires type aligned sequences and how such type aligned sequences can be used to solve the problem. Afterwards, we discuss the general solution.

4.1 A first example: tree substitution

We want to replace the implementation of the Tree data type and the substitution operator such that they have the same semantics, but better performance characteristics. Hence we will redefine the following operations:

- Observing a tree, i.e. viewing if it is a leaf or node.
- Constructing a leaf or node.
- The leaf substitution operator.

We are not concerned with other operations on trees here, they are defined in terms of the above operations.

Before we define our new data type Tree', let us start with defining what the result of observing a tree should be. Analogous to viewing a sequence data structure from the left or right, we can view a tree by observing if its root node is a leaf or a node:

```
data TreeView = Node Tree' Tree'
              | Leaf
```

Notice that the children of a Node are *not* of type TreeView, they are of the new (yet to be defined) Tree' type. To pattern match on a value of type Tree', we first need to call a function that gives the view of the Tree', i.e. a function of type:

```
toView :: Tree' → TreeView
```

This pattern is common in data abstraction [25]: it allows us to hide the implementation of the Tree' type, while still being able to pattern match on it. It is, for example, also used in efficient sequence data structures, such as the one in Data.Sequence: the pattern is used to hide the implementation of the sequence such that the user cannot differentiate between things which have multiple representation, but have the same meaning.

The Glasgow Haskell Compiler has a syntactic extension called *view patterns* which eases the usage of such data types. More precisely, it allows us to apply such a view function inside a pattern match. As an example of this, with our previous tree data type we could write a function:

```
isLeaf Leaf = True
isLeaf _    = False
```

With view patterns, this function on the new Tree' type becomes:

```
isLeaf (toView → Leaf) = True
isLeaf _               = False
```

In this way, the syntactic inconvenience of our technique is minimized.

The implementation of the Tree' data type is an explicit expression: a sequence of trees $a_0, a_1, \ldots, a_n$, such that that the result of observing such a Tree' is $a_0 \hookleftarrow a_1 \hookleftarrow \ldots \hookleftarrow a_n$.

```
newtype Tree' = Tree' (CQueue TreeView)
```

Where CQueue is an efficient sequence data structure, which we assume to be an instance of the type class for sequences defined in Figure 3(a). Very efficient purely functional sequence data structures exist: data structures where both concatenation and head/tail access run in amortized constant time [20], and even data structures where both run in worst case constant time [14, 20].

The elements of the sequence are of type TreeView, which is mutually recursive with Tree': the children of the elements in the expression are again explicit expressions. The Tree' type is a **newtype** instead of a type alias, such that we can omit the Tree' constructor from the interface, making Tree' an *abstract type*.

Constructing a leaf or node of type Tree' is then done by converting a TreeView value to a Tree' by using the following function:

```
fromView :: TreeView → Tree'
fromView x = Tree' $ singleton x
```

[4] To reduce clutter, we ignore the fact that DiffMonoid and CodensityT should actually be a **newtype** in Haskell.

The resulting tree is not (yet) an argument to the substitution operator and hence it is represented as a sequence of length one. Notice that fromView is the inverse of toView.

The implementation of the substitution operator $\hookleftarrow$ is then simply to concatenate the two explicit expressions:

```
(↩) :: Tree' → Tree' → Tree'
(Tree' l) ↩ (Tree' r) = Tree' (l ⋈̂ r)
```

Since we are using an efficient sequence data structure, this concatenation only takes (amortized) constant time.

The implementation of $\hookleftarrow$ no longer defines how to actually replace the leaves of a tree with another tree. Instead this logic is moved to the toView function, which converts an explicit expression to its view (i.e. its head normal form).

```
toView :: Tree' → TreeView
toView (Tree' s) = case viewl s of
  EmptyL → Leaf
  h ◁̂ t → case h of
      Leaf     → toView (Tree' t)
      Node l r → Node (l ↩̂ t) (r ↩̂ t)
  where (↩̂) :: Tree' → CQueue TreeView → Tree'
        (Tree' l) ↩̂ r = Tree' (l ⋈̂ r)
```

Where viewl is a function that allows us to view the sequence from the left: see if it is empty or obtain the head and tail. In contrast to continuation-passing style, converting an explicitly represented expression to an observable value does not mean converting the entire explicitly represented expression: we partially convert, keeping the children of a node as an explicit expressions.

In this way, all operations we want to support, namely construction, observation and substitution have become efficient operations. Moreover, the expressions , $(x \hookleftarrow y) \hookleftarrow z$ and $x \hookleftarrow (y \hookleftarrow z)$ lead to the same sequence, and hence performance does not depend on the association pattern. It should hence come as no surprise this approach also solves performance problems if we alternate between building trees using substitution and observing the result of such substitutions.

4.2 Solving the performance problems of generic trees using type aligned sequences

But what if we want to apply our solution to generic trees? We must then explicitly represent expressions of the form:

$$m \gg\!= f_1 \gg\!= f_2 \gg\!= f_3 \ldots \gg\!= f_n$$

The problem is that each f_i has type $a \to$ Tree b, for some a and b, and these types can *differ* between elements. This means we *cannot* use a regular sequence: to use it all elements must be of the same type.

To be able to apply our solution to such situations, we generalize sequences to *type aligned sequences*: sequences parametrized by a type constructor c, such that each element is of type c a b, for some a and b. If the last type argument to c of an element is a, then first type argument to c in the next element (if any) *must be* a. If we set the type constructor c to $(\to)$, we get type aligned sequences of functions: the output type of a function is then always the input type to the next function.

In the next section we discuss such type aligned sequences in depth and show they can be defined. For now, let us assume that we have an efficient type aligned sequence data structure called TCQueue, which is an instance of the type aligned sequence type class defined in Figure 3(b).

The elements in the sequence described above are of type $a \to$ Tree' b, for some a and b, except the first element m. We need a type constructor to describe this pattern:

```
type TreeCont a b = a → Tree' b
```

A type aligned sequence where each element is a TreeCont is then of the following type[5]:

```
type TreeCExp a b = TCQueue TreeCont a b
```

The situation is now a bit different than with our non-generic trees: an expression involving a series of binds must always start with an element of type Tree' a, whereas the rest of the elements are of type TreeCont a b, for some a and b. Hence, we implement the tree data type as explicit expression containing a first element and a sequence of right-hand-side arguments to bind.

```
data Tree' a where
  Tree' :: TreeView x → TreeCExp x a → Tree' a

data TreeView a = Leaf a | Node (Tree' a) (Tree' a)
```

This definition uses an existential type x: the first element in the expression may be a tree of any type, as long as the result of the expression is a tree containing elements of type a.

The fromView and $\hookleftarrow$ functions are adapted accordingly:

```
fromView :: TreeView a → Tree' a
fromView x = Tree' x tempty

(↩) :: Tree' a → (a → Tree' b) → Tree' b
(Tree' x s) ↩ f = Tree' x (s ⋈ tsingleton f)
```

As before, the actual logic of substitution is moved to the view function:

```
toView :: Tree' a → TreeView a
toView (Tree' b t) = case b of
    Leaf a → case tviewl t of
      TEmptyL → Leaf a
      h ◁ t   → toView ((h a) ↩̂ t)
    Node l r → Node (l ↩̂ t) (r ↩̂ t)
  where (↩̂) :: Tree a → TreeCExp a b → Tree b
        (Tree' b l) ↩̂ r = Tree' b (l ⋈ r)
```

In this way, the performance problems for any usage pattern of generic trees have also disappeared by using type aligned sequences.

4.3 The general case

Suppose we have some recursive data type X and an associative operator traversing its left argument but not its right argument. The solution is then to replace the data type X by an abstract data type X' and rewrite the problematic operator by performing the following steps:

1. Replace X with two mutually recursive data types: one for the abstract type containing the explicit expression (X') and one view type, which is the same as the original X, but the self-references have been replaced by X'.
2. Define the original operator on X' by concatenating the explicit expressions.
3. Define a fromView function that converts a view value to an X' expression by constructing an explicit expression with one element.
4. Define a toView view function that evaluates an explicit expression to its view, using the workings of the original operator.

A type aligned sequence must be used if the type of the right argument of the operator depends on the type of the left argument of the operator.

Notice that explicitly representing expressions in this way means that applying the operator with the identity element does

[5] To reduce clutter, we ignore that TreeCont must be a **newtype** for this to work in current Haskell.

```
class Sequence s where
  empty      :: s a
  singleton  :: a → s a
  (⋈̂)        :: s a → s a → s a
  viewl      :: s a → ViewL s a

data ViewL s a where
  EmptyL     :: ViewL s a
  (◁̂)        :: a → s a → ViewL s a
```

(a) A type class for regular sequences.

```
class TSequence s where
  tempty     :: s c x x
  tsingleton :: c x y → s c x y
  (⋈)        :: s c x y → s c y z → s c x z
  tviewl     :: s c x y → TViewl s c x y

data TViewl s c x y where
  TEmptyL    :: TViewl s c x x
  (◁)        :: c x y → s c y z → TViewl s c x z
```

(b) A type class for type aligned sequences.

Figure 3: Type classes for type aligned and regular sequences.

not necessarily immediately yield the original value. For example, m ≫= return and m are different *expressions*. However, we cannot observe this difference by viewing m ≫= return and m. Hence, the identity element is an identity element *up to observation*. Associativity laws directly hold, since sequence concatenation is associative. To ensure that we do not accidentally differentiate between m ≫= return and m, it is important to define the result of the above steps in an separate module and to not export the constructor of X'.

This process gives an abstract type X', with operations to construct, observe (view) and apply the operator. We argue that this resulting data type X' has the same semantics as the original data type, provided that X' is abstract. We feel that a formalization of these steps and a proof of the isomorphism of X and X' should be possible, but it is beyond the scope of this paper.

5. Type aligned sequences

In the previous section, we saw that type aligned sequences are required to explicitly represent expressions involving operators where the type of the left argument depends on the type of the right argument. We now introduce type aligned sequences, discuss their relation with regular sequences, and show an example of how a sequence data type can be converted into a type aligned sequence data type.

5.1 Definition and intuition

Type aligned sequences are best explained by an example: a type aligned sequence of *functions* is a sequence $f_1, f_2, f_3 \ldots f_n$ such that the composition of these functions $f_1 \circ f_2 \circ f_3 \circ \ldots \circ f_n$ is well typed. In other words: the result type of each function in the sequence must be the same as the argument type of the next function (if any). In general, the elements of a type aligned sequence do not have to be functions, i.e. values of type a → b, but can be values of type (c a b), for some binary type constructor c. Hence, we define a *type aligned sequence* to be a sequence of elements of the type $(c\ a_i\ b_i)$ with the side-condition $b_{i-1} = a_i$. If s is the type of a type aligned sequence data structure, then (s c a b) is the type of a type aligned sequence where the first element has type (c a x), for some x, and the last element has type (c y b), for some y.

It may be instructive to think of a type aligned sequence as a *path through a directed graph*. In this directed graph each node is a *type* and there is an edge from type a to type b *for each value* of type (c a b). Hence, we call a value of type (c a b) a c-*edge*. A type aligned sequence of type (s c a b) is then a sequence of c-edges such that they form a path from a to b trough this graph: the target of each edge is the source of the next edge.

Type aligned sequences can be defined using Generalized Algebraic Data Types (GADTs) [3]. As a simple example of this, consider a type aligned list:

```
data TList c x y where
  Nil   :: TList c x x
  (⍰)   :: c x y → TList c y z → TList c x z
```

In the graph interpretation, the empty type aligned sequence corresponds to an empty path, and hence the empty list is a path from x to x, for any x. The Cons constructor adds one c-edge to the front of a path, the types ensure that the target of this c-edge is the source of the rest the path.

5.2 Relation with regular sequences

The only difference between regular sequences and type aligned sequences are the types: TList differs from the ordinary list only in the more precise types of its constructors. In fact, type aligned sequences are a *generalization* of regular sequences: any type aligned sequence can be used as a regular sequence, but not the other way around. We can use a type aligned sequence as a regular sequence by effectively "partially erasing" the extra types with the following construction:

```
data AsUnitLoop a b c where UL :: a → AsUnitLoop a () ()
```

By using this construction, there exists an edge from () to () for each value of type a in the graph interpretation. Since there are no other edges, the graph effectively has just one node: the other types are unreachable. Hence, a regular list $a_1 : a_2 : a_3 \ldots a_n : []$ of type [a] corresponds to a type aligned list:

$$\mathsf{UL}\ a_1 \mathbin{⍰} \mathsf{UL}\ a_2 \mathbin{⍰} \mathsf{UL}\ a_3 \ldots \mathsf{UL}\ a_n \mathbin{⍰} \mathsf{Nil}$$

of type TList (AsUnitLoop a) () () . This type aligned list corresponds to a path of length *n* through the graph consisting solely of self-loops on (), where each edge corresponds to a value of type a.

We can use this construction to provide an instance for the regular sequence class (Figure 3(a)) for any instance of the type aligned sequence class (Figure 3(b)):

```
type AsSequence s a = s (AsUnitLoop a) () ()

instance TSequence s ⇒ Sequence (AsSequence s) where
  empty     = tempty
  singleton = tsingleton ∘ UL
  (⋈̂)       = (⋈)
  viewl  s  = case tviewl s of
                EmptyL   → TEmptyL
                UL h ◁ t → h ◁̂ t
```

A benefit of using type aligned sequences in this way, instead of directly using regular sequences, is that type aligned sequences rule out a class of implementation bugs: the types in a type aligned sequence enforce the ordering of the elements. Hence, accidentally switching two elements will result in a type error, as the resulting sequence may not be a path. In contrast, in regular sequences the types do not enforce the ordering of the elements and an accidental change of order in, for instance, the definition of concatenation would have gone unnoticed by the type checker.

```
data Pair c a b where
  (×) :: c a w → c w b → Pair c a b

data Buffer c a b where
  B1 :: c a b → Buffer c a b
  B2 :: Pair c a b  → Buffer c a b

data Queue c a b where
  Q0 :: Queue c a a
  Q1 :: c a b → Queue c a b
  QN :: Buffer c a x → Queue (Pair c) x y
        → Buffer c y b → Queue c a b

(|▷) :: Queue c a w → c w b → Queue c a b
q |▷ b = ...
viewl :: Queue c a b → TViewl Queue c a b
viewl q = ...
```

Figure 4: A type aligned queue data structure.

In general, sequences, i.e. words over some alphabet, are *free monoids*, whereas paths through a directed graph are *free categories* [1]. Sequences in programming languages typically are homogeneous: they require that each element has the same type. The alphabet is then the set of values of the given type. Similarly, type aligned sequences are paths through the directed graph where the edges are formed by the values of type (c a b), for all types a and b.

Indeed, any sequence data type can be made an instance of Monoid, without assuming anything about the elements of the sequence. Similarly, any type aligned sequence data type can be made an instance of Category, without assuming anything about the elements of the type aligned sequence:

```
instance Sequence s ⇒ Monoid (s a) where
  mempty  = empty
  mappend = (⋈)

instance TSequence s ⇒ Category (s c) where
  id  = tempty
  (∘) = flip (⋈)
```

The fact that we can use any type aligned sequence as a regular sequence also has a theoretical motivation: a monoid corresponds to a category with just one object, the elements in the monoid are now arrows (morphisms) from this one object to itself and the monoid operation is arrow composition [1]. Hence, a free monoid corresponds to the free category over a graph with just one node, where the self-edges correspond to the elements of the alphabet. This is exactly what we did with AsUnitLoop above: it makes every value of type a into a self-edge on the node ().

5.3 An example of making sequences type aligned: efficient queues

Generalizing the types of a sequence data type so that it becomes a type aligned sequence data type, means generalizing the constructor types, and assuring (that is, "proving" to the type checker) that all operations on the data type preserve the element order. This generalization requires some creativity but in our experience, it is a straightforward operation. In the code accompanying this paper we show type aligned versions of *finger trees* [9] and of a worst case constant time catenable queue [20, 21].

As an not entirely trivial example of turning a sequence data structure into a type aligned sequence data structure, consider the (non-catenable) queue shown in Figure 4. This data structure is essentially the same as the queue presented in Okasaki's *Purely functional Data Structures* [21, §8.4] but the types have been generalized.

To generalize this queue to a type aligned sequence data structure, we needed to generalize not only the types of the constructors of the queue, but also the types of the constructors of the pairs and buffers of which it consists. Before generalizing the types, both elements of a pair had the same type, but now the elements are c-edges such that they form a path of length two. A buffer can hold either a single element or a pair and the types of these constructors have been generalized straightforwardly. Slightly less obvious is generalizing the types of the constructors of a queue. A queue may consist of nested queues: if a queue has more than one element (constructor QN), it is represented as two buffers and a *queue of pairs*. With generalized types, the type of this queue of pairs is a type aligned queue holding (Pair c)-edges, i.e. paths of length two.

The only difference in the operations, namely en-queuing and viewing the head/tail, is their type signatures, the operations themselves are left unchanged and are hence not shown. The full code for these type aligned queues is included in the code accompanying this paper.

6. Fast Monadic Reflection

In this section we show how our solution can be used in various real-life monads. In particular, several monads offer *monadic reflection*: a way to observe, or reify, the internal state of the computation, represented in a suitable data structure. For example, the internal state of a non-determinism monad can be observed as a stream of choices. This terminology is due to Filinski [6] who modeled it after the terminology of Wand and Friedman [7]. Monadic reflection leads to alternating between building and observing, and hence leads to previously undocumented, severe performance problems. We demonstrate several examples of how we can factor out sequences in monads such that monadic reflection can be efficiently supported. In particular, we discuss LogicT transformers, iteratees (and related constructs), free monads and extensible effects.

6.1 LogicT Monad Transformers

As a first example of how we can apply our solution to a practical example, consider non-determinism monads. The MonadPlus type class extends the Monad interface with support for nondeterministic choice with backtracking. The most obvious instance of this interface is the list monad: bind is then concatMap (with the order of the arguments reversed) and mplus is concatenation. The usage of list concatenation can lead to performance problems, which can be solved by simply using a catenable queue instead.

Kiselyov, Shan, Friedman and Sabry [16] showed that a large class of logical effects, namely cut, soft cut, interleaving and fair conjunction, can all be expressed when a single function is added to the interface. This function, called msplit, essentially splits the logical computation into a computation of the first result and computation of the rest of the results. More precisely, this function has type:

```
class MonadPlus m ⇒ MonadLogic m where
  msplit :: m a → m (Maybe (a, m a))
```

It takes a logical computation and turns it into another logical computation, namely one which returns Nothing if the original logical computation had no results, and otherwise returns a Just value carrying a tuple of the first result and the logical computation of the rest of the results. This is an instance of monadic reflection: msplit allows us to observe the internal state of the monad as a stream of results. The implementation of this msplit function for lists and other sequence data structures is straightforward: it converts the empty sequence to Nothing and a non-empty sequence to a Just value of the head and tail.

```
newtype ML m a = ML { toView :: m (Maybe (a, ML m a)) }
fromView = ML
single a = return (Just (a,mzero))

instance Monad m ⇒ Monad (ML m) where
  return = fromView ∘ single
  (toView → m) ≫= f = fromView $ m ≫= λx → case x of
      Nothing    → return Nothing
      Just (h,t) → toView (f h `mplus` (t ≫= f))
  fail _ = mzero

instance Monad m ⇒ MonadPlus (ML m) where
  mzero = fromView (return Nothing)
  mplus (toView → a) b = fromView $ a ≫= λx → case x of
    Nothing    → toView b
    Just (h,t) → return (Just (h,t `mplus` b))

instance MonadTrans ML where
  lift m = fromView (m ≫= single)
instance Monad m ⇒ MonadLogic (ML m) where
  msplit (toView → m) = lift m
```

(a) Original implementation.

```
newtype ML m a = ML ( CQueue (m (Maybe (a, ML m a))))
fromView = ML ∘ singleton

instance Monad m ⇒ MonadPlus (ML m) where
  mzero = ML empty
  mplus (ML a) (ML b) = ML (a ⋈̂ b)

toView :: Monad m ⇒ ML m a → m (Maybe (a, ML m a))
toView (ML s) = case viewl s of
   EmptyL → return Nothing
   h ◁̂ t → h ≫= λx → case x of
     Nothing    → toView (ML t)
     Just (hi,ML ti) → return (Just (hi,ML $ ti ⋈̂ t))

-- the other code is unchanged
```

(b) Changes to the original implementation.

Figure 5: A stream implementation of MonadLogic

However, an efficient monad *transformer* that adds non-determinism to an arbitrary monad is not defined so easily. In a functional pearl [8], Hinze systematically derives such a non-determinism monad transformer implementation. He then notes that a left-associated mplus expression has quadratic performance, and solves this by using continuation-passing style. Note that there is no problem with bind for a non-determinism monad: like concatMap for lists, it traverses both the left argument and (the result of) the right argument. Kiselyov et al. show how the monad transformer implementation of Hinze can be adapted such that it is also an instance of MonadLogic. Although it can be really tricky to see this directly from the code, this instance of MonadLogic has severe performance problems. Effectively, their implementation of msplit corresponds to converting a difference list to a list and converting to tail of the list to a difference list again. Hence, each invocation of msplit will add one extra operation per result in the remainder of the logical computation.

Their implementation uses continuation-passing style with two continuations, but the point of this paper is that it is better to make the sequence explicit instead of representing it as a tree of functions (i.e. CPS). Hence, we do not apply our method to this implementation, but to a standard stream implementation of backtracking [26] as shown in Figure 5(a). In this implementation, the ML type is essentially a list where each node of the list is the result of a computation in the underlying monad. The list can be empty (Nothing) or a head and tail (Just (a,ML m a)). The definitions are then analogous to the definitions for the lists: mplus is concatenation and ≫= is like concatMap.

Notice that ML is *not* the same as the ListT construction:

```
newtype ListT m a = ListT { runListT :: m [a] }
instance Monad m ⇒ Monad (ListT m) where ...
```

This construction only yields a monad if the argument monad, m, is commutative [12]. The difference is that in ML each node in the "list" is the result of a computation in the underlying monad, whereas with the ListT construction the *entire* list is the result of a single computation in the underlying monad.

An example of the asymptotic performance problem is the following function which obtains at most n solutions of a logical computation.

```
seqN :: MonadLogic m ⇒ Int → m a → m [a]
seqN n m
  | n ≡ 0      = return []
  | otherwise  = msplit m ≫= λx → case x of
                   Nothing    → return []
                   Just (a,m) → liftM (a:) (seqN (n−1) m)
```

Figure 6(a)[6] shows, for different implementations, the running time of obtaining n natural numbers using seqN, where the natural numbers are defined as follows[7]:

```
nats = natsFrom 1 where
  natsFrom n = return n `mplus` natsFrom (n + 1)
```

Obtaining a number of solutions requires us to recursively split the logical computation, and hence the two continuation implementation as implemented in Hackage package LogicT has quadratic running time. Of course, this is just a micro-benchmark constructed to illustrate the problem. However, this problem does not only occur on the natural numbers: it occurs *any time* we request only some, instead of all, solutions to a logical computation. This is highly counter-intuitive: it is much faster to obtain *all* results than some results. Moreover, since we are talking about monad *transformers*, requesting all results is not always an option: it may invoke undesired and/or irrevocable effects in the underlying monad.

The same problem occurs with the interleave operator as described by Kiselyov et al., which ensures fair consideration between two branches of a logical computation. An example usage of this operator is the following the logical computation:

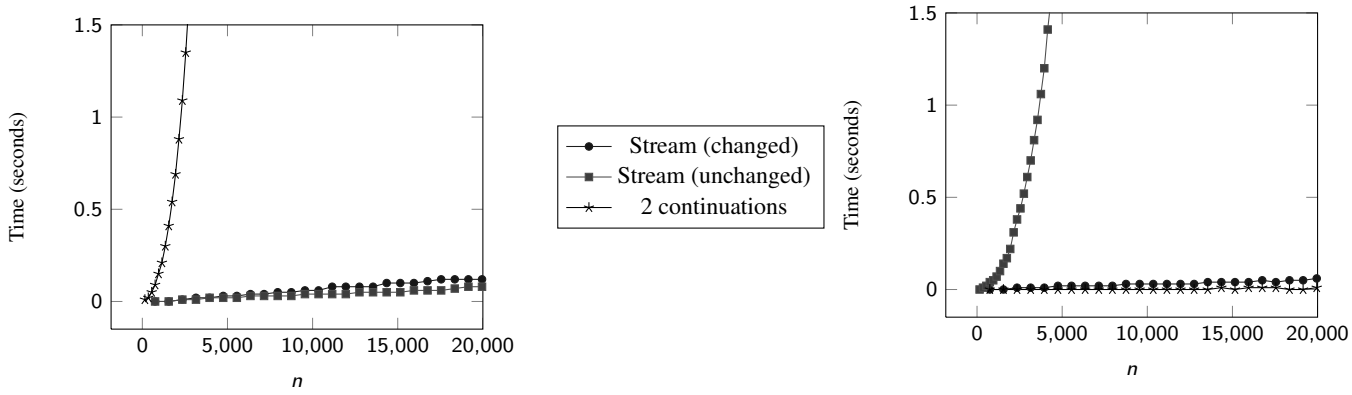
```
unfair = do x ← nats `mplus` return 0
            if x ≡ 0 then return x else mzero
```

The behavior of mplus in these implementations is that it first considers all solutions from its left argument, and only afterwards considers the solutions of its right argument. Since nats has an infinite number of results, this computation will never yield a solution. If interleave is used instead of mplus, then solutions from nats and return 0 are considered alternately and the computation will yield a solution. This interleave operator is defined in terms of mplus and msplit as follows:

```
interleave :: m a → m a → m a
interleave l r = msplit l ≫= λx → case x of
    Nothing    → r
    Just (h,t) → return h `mplus` interleave r t
```

[6] These measurements are the median of 5 runs and were performed on an AMD Phenom II X4 905e Processor CPU running Linux 3.2.0 on binaries produced with the GHC 7.6.3 (optimization level 2). The fixed stream implementation uses a worst case constant time catenable queue.

[7] (a `f` b) is an alternative notation for (f a b).



(a) Running time splitting a logical computation of natural numbers n times.

(b) Running time of observing all results in a left-associated mplus expression with n elements.

Figure 6: Running time of msplit and mplus micro benchmarks for LogicT.

Since interleave recursively splits the remaining computation of both arguments, any usage of it while using a two continuation implementation of backtracking will lead to performance problems. For instance, the following logical computation:

```
test = choose [1...n] `interleave` choose [n...1]
  where choose l = foldr mplus mzero (map return l)
```

also runs in $O(n^2)$. The same problem occurs when using using the fair conjunction operator, which is defined in terms of interleave. The cut and soft cut operators are also problematic, but much less severely: they only split the logical computation once.

Obtaining only a limited number of solutions and using the interleaving or fair conjunction operators is not problematic when using the ML implementation of MonadLogic: we can observe results directly by running a computation in the underlying monad: there is no conversion involved. Instead, the problem is now mplus: it traverses the left hand argument but not the right hand argument. Figure 6(b) shows the running time of obtaining all solutions of a left-associated mplus expression:

```
test :: MonadPlus m ⇒ Int → m Int
test n = foldl mplus mzero (map return [1...n])
```

Now the running time of the ML implementation is quadratic. The dual continuation implementation does not suffer the same problem, as it was originally derived by Hinze to solve this problem. Hence, that the performance characteristics of the ML implementation are opposite to those the two continuation implementation: the ML implementation has quadratic performance on a left-associated mplus expression, but no performance problem with msplit.

Applying our solution to the ML implementation yields the changes that are shown in Figure 5(b). The changes are very similar to the changes to the (non-generic) Tree data type: we change the ML data type to an explicit expression involving mplus, and the actual logic of non-deterministic choice is moved to the toView function. As can be seen from the graphs, after applying our method the problem with mplus disappears: the running time is now linear. Moreover, this stream implementation with our method applied to it is the *only* implementation which efficiently supports *both* msplit *and* mplus.

6.2 Iteratees and related monads

As a second example of how we can apply our solution to a practical example, consider iteratees [15]: a style of incremental input processing that overcomes the problems of lazy I/O and handle-based I/O. We consider a simplified version of iteratees where an

```
data It i a = Get (i → It i a) | Done a

instance Monad (It i) where
  return = Done
  (Ret x) ≫= g = g x
  (Get f) ≫= g = Get (f ⋙ g)

get :: It i i
get = Get return
```

Figure 7: Iteratees before applying our solution.

iteratee is a monadic computation that can request an input element, as shown in Figure 7.

An iteratee is in one of two possible states: the constructors of the It data type. If an iteratee is Done it simply carries the value it produces. If an iteratee needs an input element, it is a Get value, carrying a function that when given the input element returns the next iteratee state. A Monad instance for such iteratees is then defined straightforwardly. In this definition, the $(\ggg)$ operator is Kleisli composition ($f \ggg g = \lambda x \to f\ x \gg\!= g$) as introduced in section 2.3.

Although it can be easy to miss, the definition of the monadic bind, like its definition in the original paper, exhibits the problematic pattern: it traverses its left argument but not its right argument. It does not matter that $(\gg\!=)$ invokes itself by using function composition instead of application, this just obfuscates the problem.

As example of the performance problem is the following iteratee computation, that gets n elements from the input and then returns their sum:

```
sumInput :: Int → It Int Int
sumInput n = Get ( foldl (⋙) return ( replicate (n − 1) f))
  where f x = get ≫= return ∘ (+ x)
```

Where replicate n e is a function that creates a list of the length n, where each element is e. The sumInput function yields an expression of the form:

```
Get ((( return ⋙ f) ⋙ f) ⋙ f) ... ⋙ f)
```

Figure 8 shows that when the argument to Get is called with a new input element x, it costs $O(n)$ steps to obtain the next iteratee state:

```
Get (((( return ∘ (+ x)) ⋙ f) ⋙ f) ⋙ f) ... ⋙ f)
```

This very similar to the original expression, exhibiting the same problem. Hence, the running time of feeding this iteratee computa-

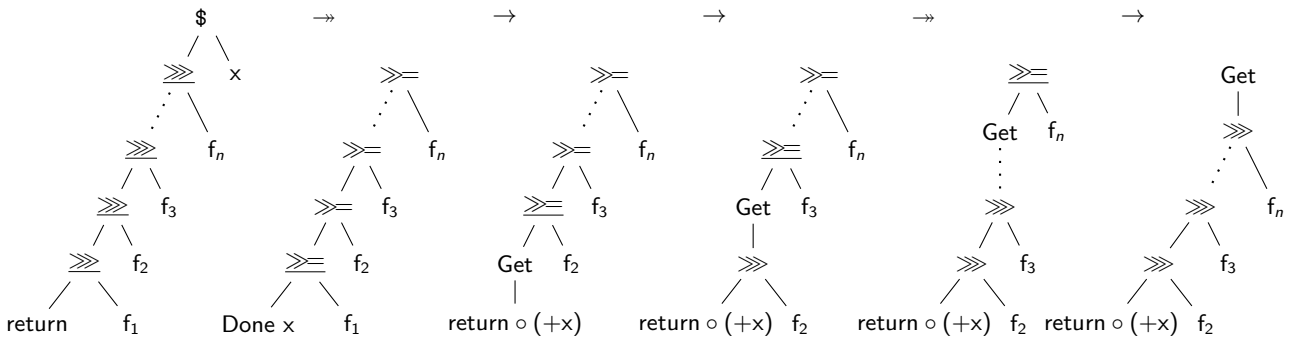


Figure 8: Example of an inefficient iteratee computation. The subscript i in f_i indicates the index of the occurrence of f.

tion n elements and obtaining their sum is quadratic. The sumInput function can easily be made to run in linear time by simply switching from foldl to foldr. However, in general solving such performance problems by avoiding the problematic pattern is not as simple: we must then make sure that that each left argument to bind cannot be the result of a bind.

We can solve the problem with repeated binds by using the codensity monad transformer, as defined in Section 3.2, as proposed by Voigtländer [24]. When using this method, we only use codensity transformed iteratees to build monadic expressions:

```
type ItCo i a = CodensityT (It i) a
```

We then redefine get so that it gives a codensity transformed iteratee:

```
getCo :: ItCo i i
getCo = rep get
```

A monadic expression built in this way will then always result in a right-associated expression when converted to a regular iteratee computation, thus avoiding the problem of repeated binds.

We now find ourselves in a familiar situation: this method makes alternating between building and observing problematic. An example of this is the following, often useful, parallel iteratee composition function, defined as a regular (non-codensity transformed) iteratee function:

```
par :: It i a → It i b → It i (It i a, It i b)
par l r
  | Done _ ← l = Done (l, r)
  | Done _ ← r = Done (l, r)
  | Get f ← l, Get g ← r = get >>= λx → par (f x) (g x)
```

This operator runs both iteratees in parallel, feeding each input element to both, until at one of the iteratees is done. Afterwards, the remaining iteratee computation of both arguments is returned, which can then be composed again with other iteratees using par and $\gg=$. The par function is an instance of monadic reflection: we observe the internal state of both iteratees.

If we want to use par on codensity transformed iteratees, we need to redefine it as follows:

```
parCo :: ItCo i a → ItCo i b
         → ItCo i (ItCo i a, ItCo i b)
parCo l r = rep (par (abs l) (abs r)) >>=
              (λ(l, r) → return (rep l, rep r))
```

We need to eliminate the codensity transformer using abs to observe the states of both iteratees. After applying the original par function, we want to be able to compose the resulting iteratees again with $\gg=$ and parCo. However, they are no longer codensity transformed iteratees, while other iteratees are in this form to avoid the problems with bind. We need to convert the rest of the resulting iteratees back to codensity transformed form. Hence, each invocation of parCo adds an extra operator per Get in the remaining iteratee, which can easily lead to performance problems when iteratees are long lived and used in many invocations of parCo.

A related construction is monadic *coroutines*, which are like iteratees except that they also output an element each time they request an input element. Blažević [2] presents an extensive library for such coroutines, but his coroutine definition suffers from the same problem as the original iteratee definition.

Another guise of the same situation occurs in monadic FRP [23]: a framework which essentially applies coroutines in a functional reactive programming (FRP) setting. In monadic FRP, a combinator very similar to par is at the heart of composing reactive computations and the bind in the paper has the same problem as the original iteratees. In fact, the motivation for this work is that we noticed that our monadic FRP program became progressively slower, due to repeated application of bind on the results of par, and eventually came to a grinding halt. Since par is used often in monadic FRP, and coroutines can live for a long time, being used in many invocations of par, the use of the codensity monad would also lead to a severe slowdown. With our solution applied, monadic FRP programs no longer become progressively slower, running efficiently no matter what the usage pattern.

Our solution can be applied to iteratees, coroutines and monadic FRP. By using an efficient type aligned sequence data structure, the performance of improves dramatically, without constraining ourselves by disallowing functions involving monadic reflection like par. We do not show the code for this due to space considerations, but instead note that iteratees, coroutines and monadic FRP are all instances of a construction known as a *free monad*, which we discuss and show the improved code of in the next section.

6.3 Free Monads

Swierstra [22] shows how a monad instance can be defined for any functor, resulting in a monad that is called the *free monad* [1] on that functor. This construction is defined as follows:

```
data FreeMonad f a = Pure a
                   | Impure (f (FreeMonad f a))

instance Functor f ⇒ Monad (FreeMonad f) where
  return = Pure
  (Pure x)   >>= f = f x
  (Impure t) >>= f = Impure (fmap (>>= f) t)
```

Swierstra then notes that several well known monads are free monads. For example, the Maybe monad is the free monad on the following functor:

```
data One a = One deriving Functor
```

Now (Pure a) corresponds to (Just a) and (Impure One) corresponds to Nothing.

However, for many functors this construction leads to asymptotic problems. Consider for example the following Functor:

```
newtype Get i a = Get (i → a) deriving Functor
```

A free monad on this functor corresponds to the iteratees we saw in the previous section. Free monads over the following functors:

```
data Node a = Node a a deriving Functor
data Yield out inn a = Yield out (inn → a) deriving Functor
```

correspond to the generic trees with substitution and coroutines, respectively. It should come as no surprise that the performance problem of iteratees, generic trees and coroutines did not go away by formulating them as free monads. Again, we could use continuation-passing style, but this would make functions like par expensive.

We solve these problem *for all* free monads by simply applying our solution. The definition of free monads then becomes:

```
type FC f a b = a → FreeMonad f b
type FMExp f a b = TCQueue (FC f) a b

data FreeMonad f a where
  FM :: FreeMonadView f x → FMExp f x a → FreeMonad f a
data FreeMonadView f a = Pure a
                       | Impure (f (FreeMonad f a))
fromView x = FM x tempty

toView :: Functor f ⇒ FreeMonad f a → FreeMonadView f a
toView (FM h t) = case h of
   Pure x →
    case tviewl t of
      TEmptyL → Pure x
      hc ◁ tc → toView (hc x ≫= tc)
   Impure f → Impure (fmap (≫= t) f) where
 (≫=) :: FreeMonad f a → FMExp f a b → FreeMonad f b
 (FM h t) ≫= r = FM h (t ⋈ r)

instance Monad (FreeMonad f) where
  return = fromView ∘ Pure
  (FM m r) ≫= f = FM m (r ⋈ singleton f)
```

Notice that this code is very similar to the code we got from applying our solution to our generic tree example in Section 4.2. This should come as no surprise: generic trees are free monads.

As usual, the code for these adapted free monads is included in the code accompanying this paper, as well as a benchmark demonstrating the performance problem and that our method solves it.

6.4 Extensible effects

Recently Kiselyov, Sabry, Swords and Foppa introduced *extensible effects* [17]: a framework for composing and implementing computational effects that overcomes the problems of monad transformers in terms of efficiency, expressiveness and ease of notation. In this framework an effect is an interaction between a client and a handler: the client sends a value describing the desired effect to the handler, which in turn executes the desired effect and passes the result to the client.

The approach of Kiselyov et al. uses functors to describe both which effect to request and how to continue afterwards. For example, both the request to modify a state and how to proceed afterwards, are represented by the following functor:

```
data ModifyState s w =
  ModState (s → s) (s → w) deriving Functor
```

The first argument tells the handler how to modify the state, whereas the second argument tells the handler how to continue afterwards, it takes the new state and then produces some w. The free monad over this functor is then the value that is interpreted by the handler: if the value is Impure (ModState f c), it applies the function f to the state and calls the function c with the new state. This may again yield an Impure value and the process continues until the handler sees a Pure value.

The *extensible* in extensible effects comes from the fact that handlers do not interpret a free monad over a single functor, but a free monad over an *open union* of functors. An open union is a value that can be of any type in a *set* of types. This distinguishes it from a closed union, for example Either a b, which has a *list* of types. Kiselyov et al. then show an implementation of an open unions of functors, which in itself is again a functor. In this way handlers for different effects can be stacked: if a handler does not handle the desired effect, the value describing the effect is passed to the next handler in the stack.

However, as we saw in the previous section, many functors give rise to performance problems when using a (non-adapted) free monad. For functors describing effects, this is the case if the effect produces some result which is then passed to a continuation function. This is always the case, except for exceptions.

Kiselyov et al. avoid this problem by using a variant of free monads using continuation-passing style. This has the advantage that it avoids the performance problems of wrong groupings of expressions involving bind, but it has the disadvantage that handlers must be written in continuation-passing style. In a related paper, Kammar et al. [13] avoid the performance problem by (implicitly) applying the codensity monad.

Both approaches lead to performance problems when effects requiring reflection such as iteratees, LogicT transformers or delimited continuations are modeled. With our solution, extensible effects can directly be expressed as (adapted) free monads over open unions, without the need for manual continuation-passing style or the codensity monad. Moreover, effects that require reflection *can* then be efficiently supported. An example implementation of extensible effects as efficient free monads is included in the code accompanying this paper, as well as a benchmark involving reflection in the form of a logical cut effect, that is quadratic in the original implementation, but linear in our adapted implementation.

7. Conclusion

Associative operators that traverse their left argument, but not their right argument, can lead to asymptotic overhead. A popular cure is to use continuation-passing style, but this cure is only effective if our usage is strictly separated into a build and an observation phase, otherwise the cure is as bad as the disease.

We presented a solution that solves such performance problems for any usage pattern, even when alternating between building and observing. Our solution reveals a hidden sequence, namely repeated applications of such a problematic operator, and makes it concrete using an efficient sequence data structure.

To support operators where the type of the right argument depends on the type of the left argument, such as the monadic bind, we introduced a generalization of sequences called type aligned sequences. Type aligned sequences enforce the ordering of their elements, and hence rule out ordering bugs.

Monadic reflection, i.e. a way to observe, or reify, the internal state of a monadic computation requires us to alternate between building and observing. We showed that reflection does not have to lead to remorse: our solution efficiently supports reflection. We have demonstrated that our solution can yield an asymptotic running time improvement in iteratees (and related constructs), LogicT transformers, free monads and extensible effects.

Our solution is not limited to the examples we discussed in this paper. In the accompanying code, we show how sequences can be factored out in delimited continuations [5] and term monads [19]. Given the simplicity of the problematic pattern and the widespread usage of continuation-passing style, we suspect that there are many more applications of our solution hiding in corners where we have not looked yet.

Acknowledgment

We thank Jan Rutten, Koen Claessen and the anonymous reviewers for helpful discussions and comments on this paper.

References

[1] S. Awodey. *Category theory*. Oxford University Press, 2006.

[2] M. Blažević. Coroutine pipelines. *The Monad Reader*, 19:29–50, 2011.

[3] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[4] K. Claessen. Functional pearl: Parallel parsing processes. *J. of Functional Programming*, 14:741–757, 2004.

[5] R. K. Dyvbig, S. Peyton Jones, and A. Sabry. A monadic framework for delimited continuations. *J. of Functional Programming*, 17(6): 687–730, 2007.

[6] A. Filinski. Representing monads. In *Proc. of the 21th Symposium on Principles of Programming Languages*, pages 446–457, 1994.

[7] D. P. Friedman. The mystery of the tower revealed: A nonreflective description of the reflective tower. *LISP and Symbolic Computation*, 1(1):298–307, 1988.

[8] R. Hinze. Deriving backtracking monad transformers. In *Proc. of the 5th International Conference on Functional Programming*, pages 186–197, 2000.

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

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

[11] M. Jaskelioff. Modular monad transformers. In *Transactions on Programming Languages and Systems*, pages 64–79, 2009.

[12] M. P. Jones and L. Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University, 1993.

[13] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proc. of the 18th International Conference on Functional Programming*, 2013.

[14] H. Kaplan and R. E. Tarjan. Purely functional, real-time deques with catenation. *J. of the ACM*, 46(5):577–603, 1999.

[15] O. Kiselyov. Iteratees. In *Proc. of the 11th International Symposium on Functional and Logic Programming*, pages 166–181, 2012.

[16] O. Kiselyov, C. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers (functional pearl). In *Proc. of the 10th International Conference on Functional Programming*, pages 192–203, 2005.

[17] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: An alternative to monad transformers. In *Proc. of the '13 Symposium on Haskell*, pages 59–70, 2013.

[18] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proc. of the 22nd Symposium on Principles of Programming Languages*, pages 333–343, 1995.

[19] C. Lin. Programming monads operationally with unimo. In *Proc. of the 11th International Conference on Functional Programming*, pages 274–285, 2006.

[20] C. Okasaki. Simple and efficient purely functional queues and deques. *J. of Functional Programming*, 5:583–592, 1995.

[21] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

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

[23] A. van der Ploeg. Monadic functional reactive programming. In *Proc. of the '13 Symposium on Haskell*, pages 117–128, 2013.

[24] J. Voigtländer. Asymptotic improvement of computations over free monads. In *Proc. of the 9th International Conference on Mathematics of Program Construction*, pages 388–403, 2008.

[25] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proc. of the 14th Symposium on Principles of Programming Languages*, pages 307–313, 1987.

[26] M. Wand and D. Vaillancourt. Relating models of backtracking. In *Proc. of the 9th International Conference on Functional Programming*, pages 54–65, 2004.