Asymptotic Improvement of Computations over Free Monads*

Janis Voigtländer

Institut für Theoretische Informatik
Technische Universität Dresden
01062 Dresden, Germany

janis.voigtlaender@acm.org

Abstract. We present a low-effort program transformation to improve the efficiency of computations over free monads in Haskell. The development is calculational and carried out in a generic setting, thus applying to a variety of datatypes. An important aspect of our approach is the utilisation of type class mechanisms to make the transformation as transparent as possible, requiring no restructuring of code at all. There is also no extra support necessary from the compiler (apart from an up-to-date type checker). Despite this simplicity of use, our technique is able to achieve true asymptotic runtime improvements. We demonstrate this by examples for which the complexity is reduced from quadratic to linear.

1 Introduction

Monads [1] have become everyday structures for Haskell programmers to work with. Not only do monads allow to safely encapsulate impure features of the programming language [2,3], but they are also used in pure code to separate concerns and provide modular design [4,5]. But, as usual in software construction, modularity comes at a cost, typically with respect to program efficiency. We propose a method to improve the efficiency of code over a large variety of monads. A distinctive feature is that this method is non-intrusive: it preserves the appearance of code, with the obvious software engineering benefits.

Since our approach is best introduced by considering a concrete example, illustrating both the problem we address and our key ideas, that is exactly what we do in the next section. Thereafter, Sect. 3 develops the approach formally, embracing a generic programming style. Further example material is provided in Sects. 4 and 5, where the latter emphasises comparison to related work, before Sect. 6 concludes.

The code that we present throughout requires some extensions over the Haskell 98 standard, in particular rank-2 polymorphism and multi-parameter type constructor classes. It was tested against both GHC (version 6.6, flag `-fglasgow-exts`) and Hugs (version of March 2005, flag `-98`), and is available online at `http://wwwtcs.inf.tu-dresden.de/~voigt/Improve.lhs`.

* In *MPC 2008, Proc.*, volume 5133 of LNCS, pages 388–403.

2 A Specific Example

We first study a simple and somewhat artificial example of the kind of transformation we want to achieve. This prepares the ground for the more generic development in the next section, and more practical examples later on.

Consider the following datatype of binary, leaf-labelled trees:

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

An important operation on such trees is substituting leaves by trees depending on their labels, defined by structural induction as follows:

```
subst :: Tree α → (α → Tree β) → Tree β
subst (Leaf a) k = k a
subst (Node t1 t2) k = Node (subst t1 k) (subst t2 k)
```

Note that the type of labels might change during such a substitution.

It is well-known that trees with substitution form a monad. That is,

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

defines an instance of the following type constructor class:

```
class Monad μ where
  return :: α → μ α
  (>>=) :: μ α → (α → μ β) → μ β
```

where the following three laws hold:

$$(\mathsf{return}\ a \gg\!= k) = k\ a \tag{1}$$

$$(m \gg\!= \mathsf{return}) = m \tag{2}$$

$$((m \gg\!= k) \gg\!= h) = (m \gg\!= (\lambda a \to k\ a \gg\!= h)) \tag{3}$$

An example use of the monad instance given above is the following program generating trees like those in Fig. 1:

```
fullTree :: Int → Tree Int
fullTree 1 = Leaf 1
fullTree (n+1) =
  do
    i ← fullTree n
    Node (Leaf (n−i)) (Leaf (i+1))
```

Note that the second equation is equivalent to

```
fullTree (n+1) = fullTree n >>= λi → Node (Leaf (n−i)) (Leaf (i+1))
```

and thus to

```
fullTree (n+1) = subst (fullTree n) (λi → Node (Leaf (n−i)) (Leaf (i+1)))
```

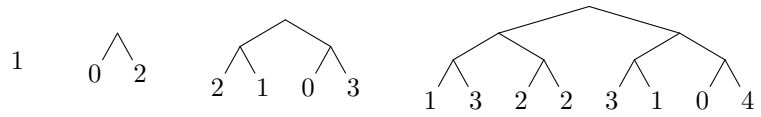

Fig. 1. fullTree 1, fullTree 2, fullTree 3, fullTree 4

This means that to create, for example, the tree fullTree 4, the following expression is eventually evaluated:

$$\mathsf{subst}\ (\mathsf{subst}\ (\mathsf{subst}\ (\textsc{Leaf}\ 1)\ \cdots)\ \cdots)\ \cdots \tag{4}$$

The nested calls to subst mean that prefix fragments of the overall tree structure will be traversed again and again.

In general, the asymptotic time complexity of computing fullTree n is of the order 2^n, which is no surprise as the size of the finally computed output is of that order as well. But a more interesting effect occurs when that output is only partially demanded, such as by the following function:

```
zigzag :: Tree Int → Int
zigzag = zig
  where
    zig (Leaf n) = n
    zig (Node t1 t2) = zag t1
    zag (Leaf n) = n
    zag (Node t1 t2) = zig t2
```

Now the expression

$$\mathsf{zigzag}\ (\mathsf{fullTree}\ n) \tag{5}$$

has *quadratic* runtime in n despite exploring only a single tree path of length *linear* in n. To see where the quadratic runtime comes from, consider the following partial reduction sequence for zigzag (fullTree 4), starting from (4) and having reordered a bit the lazy evaluation steps by first, and only, recording those involving subst:

$$\begin{aligned}
&\mathsf{zigzag}\ (\mathsf{subst}\ (\mathsf{subst}\ (\mathsf{subst}\ (\textsc{Leaf}\ 1)\ \cdots)\ \cdots)\ \cdots)\\
\Rightarrow\ &\mathsf{zigzag}\ (\mathsf{subst}\ (\mathsf{subst}\ (\textsc{Node}\ (\textsc{Leaf}\ 0)\ (\textsc{Leaf}\ 2))\ \cdots)\ \cdots)\\
\Rightarrow^2\ &\mathsf{zigzag}\ (\mathsf{subst}\ (\textsc{Node}\ (\textsc{Node}\ (\textsc{Leaf}\ 2)\ (\textsc{Leaf}\ 1))\ (\mathsf{subst}\ (\textsc{Leaf}\ 2)\ \cdots))\ \cdots)\\
\Rightarrow^3\ &\mathsf{zigzag}\ (\textsc{Node}\ (\textsc{Node}\ (\mathsf{subst}\ (\textsc{Leaf}\ 2)\ \cdots)\ (\textsc{Node}\ (\textsc{Leaf}\ 2)\ (\textsc{Leaf}\ 2)))\ \cdots)\\
\Rightarrow^*\ &\cdots
\end{aligned}$$

The challenge is to bring the overall runtime for (5) down to linear, but to do so without changing the structure of the code for fullTree.

The situation here is similar to that in the well-known definition of naïve list reversal, where left-associatively nested appends cause quadratic runtime. And in fact there is a similar cure. We can create an alternative representation of trees somewhat akin to the "novel representation of lists" of Hughes [6], also

known as difference lists. Just as the latter abstract over the end of a list, we abstract over the leaves of a tree as follows:

```
newtype CTree α = CTree (∀β. (α → Tree β) → Tree β)
```

The connection between ordinary trees and their alternative representation is established by the following two functions:

```
rep :: Tree α → CTree α
rep t = CTree (subst t)

abs :: CTree α → Tree α
abs (CTree p) = p Leaf
```

We easily have abs ∘ rep = id. Moreover, the representation type forms itself a monad as follows:

```
instance Monad CTree where
    return a = CTree (λh → h a)
    CTree p >>= k = CTree (λh → p (λa → case k a of CTree q → q h))
```

But to use it as a drop-in replacement for Tree in the definition of fullTree, the type constructor CTree need not only support the monad operations, but also the actual construction of (representations of) non-leaf trees. To capture this requirement, we introduce the following type constructor class:

```
class Monad μ ⇒ TreeLike μ where
    node :: μ α → μ α → μ α
```

For ordinary trees, the instance definition is trivial:

```
instance TreeLike Tree where
    node = Node
```

For the alternative representation, we have to take care to propagate the abstracted-over leaf replacement function appropriately. This is achieved as follows:

```
instance TreeLike CTree where
    node (CTree p1) (CTree p2) = CTree (λh → Node (p1 h) (p2 h))
```

For convenience, we also define an abstract version of the Leaf constructor.

```
leaf :: TreeLike μ ⇒ α → μ α
leaf = return
```

Now, we can easily give a variant of fullTree that is independent of the choice of trees to work with.

```
fullTree′ :: TreeLike μ ⇒ Int → μ Int
fullTree′ 1 = leaf 1
fullTree′ (n+1) =
    do
      i ← fullTree′ n
      node (leaf (n−i)) (leaf (i+1))
```

Note that the code structure is exactly as before. Moreover,

$$\mathsf{zigzag}\ (\mathsf{fullTree}'\ n) \tag{6}$$

still needs quadratic runtime. Indeed, GHC 6.6 with optimisation settings produces exactly the same compiled code for (5) and (6). Nothing magical has happened yet: any overhead related to the type class abstraction in (6) is simply optimised away. So there appears to be neither a gain from, nor a penalty for switching from fullTree to fullTree′. Why the (however small) effort, then?

The point is that we can now switch to an asymptotically more efficient version with almost zero effort. It is as simple as writing

$$\mathsf{zigzag}\ (\mathsf{improve}\ (\mathsf{fullTree}'\ n))\,, \tag{7}$$

where all the "magic" lies with the following function:

```
improve :: (∀μ. TreeLike μ ⇒ μ α) → Tree α
improve m = abs m
```

In contrast to (5) and (6), evaluation of (7) has runtime only linear in n.

The rationale for the type of improve, as well as the correctness of the above transformation in the sense that (7) always computes the same output as (6), will be discussed in the next section, all for a more general setting than the specific type of trees and the example considered here.

We end the current section by pointing out that (7) is compiled (again by GHC 6.6) to code corresponding to

$$\mathsf{zigzag}\ (\mathsf{fullTree}''\ n\ \mathsf{Leaf})\,,$$

where:

```
fullTree'' :: Int → (Int → Tree β) → Tree β
fullTree'' 1 h = h 1
fullTree'' (n+1) h = fullTree'' n (λi → Node (h (n−i)) (h (i+1)))
```

This should make apparent why the runtime is now only of linear complexity.

3 The Generic Setting

To deal with a variety of different datatypes in one stroke, we use the by now folklore approach of two-level types [7,8].

A *functor* is an instance of the following type constructor class:

```
class Functor φ where
    fmap :: (α → β) → φ α → φ β
```

satisfying the following two laws:

$$\mathsf{fmap}\ \mathsf{id}\ t = t \tag{8}$$

$$\mathsf{fmap}\ f\ (\mathsf{fmap}\ g\ t) = \mathsf{fmap}\ (f \circ g)\ t \tag{9}$$

Given such an instance, the corresponding *free monad* (capturing terms containing variables, along with a substitution operation) is defined as follows:

```
data Free φ α = Return α | Wrap (φ (Free φ α))

instance Functor φ ⇒ Monad (Free φ) where
    return = Return
    Return a >>= k = k a
    Wrap t >>= k = Wrap (fmap (>>= k) t)
```

Of course, we want to be sure that the laws (1)–(3) hold for the instance just defined. While law (1) is obvious from the definitions, the other two require fixpoint induction and laws (8) and (9).

As an example, consider the following functor:

```
data F β = N β β

instance Functor F where
    fmap h (N x y) = N (h x) (h y)
```

Then Free F corresponds to Tree from Sect. 2, and the monad instances agree.

Back to the generic setting. What was abstraction over leaves in the previous section, now becomes abstraction over the return method of a monad. This abstraction is actually possible for arbitrary, rather than only for free monads. The straight-forward definitions are as follows:

```
newtype C μ α = C (∀β. (α → μ β) → μ β)

rep :: Monad μ ⇒ μ α → C μ α
rep m = C (m >>=)

abs :: Monad μ ⇒ C μ α → μ α
abs (C p) = p return

instance Monad (C μ) where
    return a = C (λh → h a)
    C p >>= k = C (λh → p (λa → case k a of C q → q h))
```

Even though the monad laws do hold for the latter instance, we will not need this fact later on. What we will need, however, is the abs ∘ rep = id property:

```
abs (rep m)
  = by definition of rep
abs (C (m >>=))
  = by definition of abs                                   (10)
m >>= return
  = by law (2) for the instance Monad μ
m
```

We also need to establish connections between the methods of the instances MONAD μ and MONAD (C μ). For return, we have:

$$
\begin{array}{l}
\mathsf{rep}\ (\mathsf{return}\ a) \\
\quad = \text{ by definition of } \mathsf{rep} \\
\mathsf{C}\ (\mathsf{return}\ a \gg\!=) \\
\quad = \text{ by definition of sectioning} \\
\mathsf{C}\ (\lambda h \to \mathsf{return}\ a \gg\!= h) \\
\quad = \text{ by law (1) for the instance MONAD } \mu \\
\mathsf{C}\ (\lambda h \to h\ a) \\
\quad = \text{ by definition of return for the instance MONAD (C } \mu) \\
\mathsf{return}\ a
\end{array}
\tag{11}
$$

Note that the occurrences of return in the first few lines refer to the instance MONAD μ, whereas the return in the last line lives in the instance MONAD (C μ). For the other method of the MONAD class, we get the following distribution-like property:

$$
\begin{array}{l}
\mathsf{rep}\ (m \gg\!= k) \\
\quad = \text{ by definition of } \mathsf{rep} \\
\mathsf{C}\ ((m \gg\!= k) \gg\!=) \\
\quad = \text{ by definition of sectioning} \\
\mathsf{C}\ (\lambda h \to (m \gg\!= k) \gg\!= h) \\
\quad = \text{ by law (3) for the instance MONAD } \mu \\
\mathsf{C}\ (\lambda h \to m \gg\!= (\lambda a \to k\ a \gg\!= h)) \\
\quad = \text{ by case-of-known} \\
\mathsf{C}\ (\lambda h \to m \gg\!= (\lambda a \to \mathbf{case}\ \mathsf{C}\ (k\ a \gg\!=)\ \mathbf{of}\ \mathsf{C}\ q \to q\ h)) \\
\quad = \text{ by definition of } \mathsf{rep} \\
\mathsf{C}\ (\lambda h \to m \gg\!= (\lambda a \to \mathbf{case}\ \mathsf{rep}\ (k\ a)\ \mathbf{of}\ \mathsf{C}\ q \to q\ h)) \\
\quad = \text{ by definition of } \gg\!= \text{ for the instance MONAD (C } \mu) \\
\mathsf{C}\ (m \gg\!=) \gg\!= (\mathsf{rep} \circ k) \\
\quad = \text{ by definition of } \mathsf{rep} \\
\mathsf{rep}\ m \gg\!= (\mathsf{rep} \circ k)
\end{array}
\tag{12}
$$

Next, we need support for expressing the construction of non-return values in both monads FREE ϕ and C (FREE ϕ). To this end, we introduce the following multi-parameter type constructor class:

```
class (Functor φ, Monad μ) ⇒ FreeLike φ μ where
    wrap :: φ (μ α) → μ α
```

As in Sect. 2, one instance definition is trivial:

```
instance Functor φ ⇒ FreeLike φ (Free φ) where
    wrap = Wrap
```

The other one takes a bit more thinking, but will ultimately be justified by the succeeding calculations.

```
instance FreeLike φ μ ⇒ FreeLike φ (C μ) where
    wrap t = C (λh → wrap (fmap (λ(C p) → p h) t))
```

Similarly as for the monad methods before, we would like to prove distribution of rep over wrap, thus establishing a connection between instances FreeLike ϕ μ and FreeLike ϕ (C μ). More specifically, we expect rep (wrap t) = wrap (fmap rep t). However, a straightforward calculation from both sides gets stuck somewhere in the middle as follows:

```
rep (wrap t)
  = by definition of rep
C (wrap t >>=)
  = by definition of sectioning
C (λh → wrap t >>= h)
  = by ???
C (λh → wrap (fmap (>>= h) t))
  = by definition of sectioning
C (λh → wrap (fmap (λm → m >>= h) t))
  = by case-of-known
C (λh → wrap (fmap (λm → (λ(C p) → p h) (C (m >>=))) t))
  = by definition of rep
C (λh → wrap (fmap (λm → (λ(C p) → p h) (rep m)) t))
  = by law (9) for the instance Functor φ
C (λh → wrap (fmap (λ(C p) → p h) (fmap rep t)))
  = by definition of wrap for the instance FreeLike φ (C μ)
wrap (fmap rep t)
```

On reflection, this is not so surprising, since it was to be expected that at some point we really need to consider the more specific Free ϕ versus C (Free ϕ) rather than the more general (and thus less informative) μ versus C μ as done for (10)–(12). Here now this point has come, and indeed we can reason for t of type ϕ (Free ϕ α) as follows:

```
rep (wrap t)
  = as above
C (λh → wrap t >>= h)
  = by definition of wrap for the instance FreeLike φ (Free φ)
C (λh → Wrap t >>= h)
  = by definition of >>= for the instance Monad (Free φ)          (13)
C (λh → Wrap (fmap (>>= h) t))
  = by definition of wrap for the instance FreeLike φ (Free φ)
C (λh → wrap (fmap (>>= h) t))
  = as above
wrap (fmap rep t)
```

Our "magic function" is again the same as abs up to typing:

$$\begin{aligned}&\mathsf{improve} :: \textsc{Functor}\ \phi \Rightarrow (\forall \mu.\ \textsc{FreeLike}\ \phi\ \mu \Rightarrow \mu\ \alpha) \to \textsc{Free}\ \phi\ \alpha\\ &\mathsf{improve}\ m = \mathsf{abs}\ m\end{aligned}$$

In fact, comparing their types should be instructive. Recall that

$$\mathsf{abs} :: \textsc{Monad}\ \mu \Rightarrow \mathsf{C}\ \mu\ \alpha \to \mu\ \alpha\,.$$

This type is different from that of improve in two ways. The first, and less essential, one is that abs is typed with respect to an arbitrary monad μ, whereas ultimately we want to consider the more specific case of monads of the form $\textsc{Free}\ \phi$. Of course, by simple specialisation, abs admits the following type as well:

$$\mathsf{abs} :: \textsc{Functor}\ \phi \Rightarrow \mathsf{C}\ (\textsc{Free}\ \phi)\ \alpha \to \textsc{Free}\ \phi\ \alpha\,.$$

But, more essentially, the input type of improve is $\forall \mu.\ \textsc{FreeLike}\ \phi\ \mu \Rightarrow \mu\ \alpha$, which puts stronger requirements on the argument m than just $\mathsf{C}\ (\textsc{Free}\ \phi)\ \alpha$ would do. And that is what finally enables us to establish the correctness of adding improve at will wherever the type checker allows doing so. The reasoning, in brief, is as follows:

$$\begin{aligned}&\mathsf{improve}\ m\\ &\quad = \text{ by definition of improve}\\ &\mathsf{abs}\ m\\ &\quad = \text{ by (11)–(13)}\\ &\mathsf{abs}\ (\mathsf{rep}\ m)\\ &\quad = \text{ by (10)}\\ &m\end{aligned}$$

To understand in more detail what is going on here, it is particularly helpful to examine the type changes that m undergoes in the above calculation.

1. In the first line, m has the type $\forall \mu.\ \textsc{FreeLike}\ \phi\ \mu \Rightarrow \mu\ \alpha$ (for some fixed instance $\textsc{Functor}\ \phi$ and some fixed α), because that is what the type of improve forces it to be.
2. In the second line, m has the type $\mathsf{C}\ (\textsc{Free}\ \phi)\ \alpha$, because that is what the type of abs forces it to be, taking into account that the overall expression in each line must have the type $\textsc{Free}\ \phi\ \alpha$. When going from left to right in the definition of improve, the type of m is thus specialised from $\forall \mu.\ \textsc{FreeLike}\ \phi\ \mu \Rightarrow \mu\ \alpha$ to $\mathsf{C}\ (\textsc{Free}\ \phi)\ \alpha$. This is possible, and done silently (by the type checker), since an instance $\textsc{FreeLike}\ \phi\ (\mathsf{C}\ (\textsc{Free}\ \phi))$ follows from the existing instance declarations $\textsc{Functor}\ \phi \Rightarrow \textsc{FreeLike}\ \phi\ (\textsc{Free}\ \phi)$ and $\textsc{FreeLike}\ \phi\ \mu \Rightarrow \textsc{FreeLike}\ \phi\ (\mathsf{C}\ \mu)$.
3. In the third line, m has the type $\textsc{Free}\ \phi\ \alpha$, because that is what the types of abs and rep force it to be. That type is an alternative specialisation of the original type $\forall \mu.\ \textsc{FreeLike}\ \phi\ \mu \Rightarrow \mu\ \alpha$ of m, possible due to the instance declaration $\textsc{Functor}\ \phi \Rightarrow \textsc{FreeLike}\ \phi\ (\textsc{Free}\ \phi)$. The key observation about the second versus third lines is that even though m has been type-specialised in two different ways, the *definition* (or value) of m is still the same as in the first

line. And since there it has the very general type $\forall\mu.$ FreeLike $\phi\ \mu \Rightarrow \mu\ \alpha$, we know that m cannot be built from any μ-related operations for any specific μ. Rather, its μ-structure must be made up from the overloaded operations return, $\gg\!=$, and wrap only. And since rep distributes over all of these by (11)–(13), we have

$$\mathsf{rep}\ (m :: \mathsf{Free}\ \phi\ \alpha) = (m :: \mathsf{C}\ (\mathsf{Free}\ \phi)\ \alpha)$$

A more formal proof would require techniques akin to those used for deriving so-called *free theorems* [9,10].

4. In the fourth line, m still has the type Free $\phi\ \alpha$.

The essence of all the above is that improve m can be used wherever a value of type Free $\phi\ \alpha$ is expected, but that m itself must (also) have the more general type $\forall\mu.$ FreeLike $\phi\ \mu \Rightarrow \mu\ \alpha$, and that then improve m is equivalent to just m, appropriately type-specialised. Or, put differently, wherever we have a value of type Free $\phi\ \alpha$ which is constructed in a sufficiently abstract way (via the overloaded operators return, $\gg\!=$, and wrap) that it could also be given the type $\forall\mu.$ FreeLike $\phi\ \mu \Rightarrow \mu\ \alpha$, we can apply improve to that value without changing program semantics. Yet another perspective is that improve is simply a type conversion function that can replace an otherwise anyway, but silently, performed type specialisation and has the nice side effect of potentially improving the asymptotic runtime of a program (when left-associatively arranged calls to $\gg\!=$ cause quadratic overhead).

Having studied the generic setting, we can once more return to the specific example from Sect. 2. As already mentioned, the functor F given earlier in the current section yields Free F corresponding to Tree. Moreover, in light of the further general definitions from the current section, F and its functor instance definition also give us all the remaining ingredients of our improvement approach for binary, leaf-labelled trees. In particular, the type constructor C (Free F) corresponds to CTree, and FreeLike F takes the role of TreeLike. There is no need to provide any further definitions, since all the type constructor class instances that are needed are automatically obtained from the mentioned single one, and our generic definition of improve is similarly covering the earlier more specific one. In the next sections we will benefit from this genericity repeatedly.

4 A More Realistic Example

Swierstra and Altenkirch [11] build a pure model of Haskell's teletype IO, with the aim of enabling equational reasoning and automated testing. The monad they use for this corresponds to Free F_IO for the following functor:

```
data F_IO β = GetChar (Char → β) | PutChar Char β

instance Functor F_IO where
  fmap h (GetChar f) = GetChar (h ∘ f)
  fmap h (PutChar c x) = PutChar c (h x)
```

They then provide replacements of Haskell's getChar/putChar functions that produce pure values of this modelling type rather than doing actual IO. We can do so as well, catching up to List. 1 of [11].

```
getChar :: FreeLike F_IO μ ⇒ μ Char
getChar = wrap (GetChar return)

putChar :: FreeLike F_IO μ ⇒ Char → μ ()
putChar c = wrap (PutChar c (return ()))
```

The only differences of note are the more general return types of our versions of getChar and putChar. Just as the original function versions, our versions can be used to specify any interaction. For example, we can express the following computation:

```
revEcho :: FreeLike F_IO μ ⇒ μ ()
revEcho =
  do
    c ← getChar
    when (c ≠ ' ') $
      do
        revEcho
        putChar c
```

Run against the standard Haskell definitions of getChar and putChar (and obviously, then, with the different type signature revEcho :: IO ()), the above code reads characters from the input until a space is encountered, after which the sequence just read is written to the output in reverse order.

The point of Swierstra and Altenkirch's approach is to run the very same code against the pure model instead. Computing its (or similar functions') behaviour is done by a semantics they provide in their List. 2 and which is virtually replicated here (the only differences being two occurrences of Wrap):

```
data Output α = Read (Output α) | Print Char (Output α) | Finish α

data Stream α = Cons {hd :: α, tl :: Stream α}

run :: Free F_IO α → Stream Char → Output α
run (Return a) cs = Finish a
run (Wrap (GetChar f)) cs = Read (run (f (hd cs)) (tl cs))
run (Wrap (PutChar c p)) cs = Print c (run p cs)
```

Simulating a run of revEcho on some input stream, or indeed using QuickCheck [12] to analyse many such runs, takes the following form:

```
run revEcho stream .
```

It turns out that this requires runtime quadratic in the number of characters in *stream* before the first occurrence of a space. This holds both with our definitions and with those of [11]. So these two sets of definitions are not only equivalent

with respect to the pure models and associated semantics they provide, but also in terms of efficiency. The neat twist in our setting, however, is that we can simply write

$$\mathsf{run}\ (\mathsf{improve}\ \mathsf{revEcho})\ \mathit{stream} \tag{14}$$

to reduce the complexity from quadratic to linear. The manner in which the quadraticity vanishes here is actually very similar to that observed for the "zigzag after fullTree" example at the end of Sect. 2, so we refrain from giving the code to which (14) is eventually compiled.

It is worth pointing out that the nicely general type of revEcho that makes all this possible could be automatically inferred from the function body if it were not for Haskell's dreaded monomorphism restriction. In fact, in GHC 6.6 we have the option of suppressing that restriction, in which case we need not provide the signature revEcho :: FreeLike F_IO $\mu \Rightarrow \mu$ (), and thus need not even be aware of whether we program against the pure teletype IO model in the incarnation of [11], our "magically improvable" variant of it, or indeed the standard Haskell IO monad.

5 Related Work

In this section we relate our work to two other strands of recent work that use two-level types in connection with monadic datatypes.

5.1 Structuring Haskell IO by Combining Free Monads

We have already mentioned the work by Swierstra and Altenkirch [11] on building pure models of (parts of) the Haskell IO monad. Apart from teletype IO, they also consider mutable state and concurrency. In both cases, the modelling type is a free monad and thus amenable to our improvement method. In a recent pearl [13, Sect. 7], Swierstra takes the modelling approach a step further. The free monad structure is used to combine models for different aspects of Haskell IO, and the models are not just used for reasoning and testing in a pure setting, but also for actual effectful execution. The idea is that the types derived for terms over the pure models are informative about just which kinds of effects can occur during eventual execution. Clearly, there is an interpretative overhead here, and somewhat startlingly this even affects the asymptotic complexity of programs.

For example, for teletype IO the required execution function looks as follows, referring to the original, effectful versions of getChar and putChar:

```
exec :: Free F_IO α → IO α
exec (Return a) = return a
exec (Wrap (GetChar f)) = Prelude.getChar >>= (exec ∘ f)
exec (Wrap (PutChar c p)) = Prelude.putChar c >> exec p
```

Now, main = exec revEcho unfortunately has quadratic runtime behaviour, very evident already via simple experiments with piping to the compiled version a

text file with a few thousand initial non-spaces. This is in stark contrast to running revEcho (with alternative type signature revEcho :: IO ()) directly against the IO monad. Quite nicely, simply using main = exec (improve revEcho) recovers the linear behaviour as well. So thanks to our improvement method for free monads, which is orthogonal to Swierstra's "combination by coproducts" approach, we can have it both: pure modelling with informative types and efficient execution without (too big) interpretative overhead. Of course, our improvement also works for other cases of Swierstra's approach, such as his calculator example in Sect. 6. Up to compatibility with Agda's dependent type system, it should also apply to the models Swierstra and Altenkirch provide in [14] for computation on (distributed) arrays, and should reap the same benefits there.

5.2 Short Cut Fusion for Monadic Computations

Ghani et al. [15,16] observe that the augment combinator known from work on short cut fusion [17,18] has a monadic interpretation, and thus enables fusion for programs on certain monadic datatypes. This strand of work is thus the one most closely related to ours, since it also aims to improve the efficiency of monadic computations. An immediate difference is that Ghani et al.'s transformation can at best achieve a linear speedup, but no improvement of asymptotic complexity. More specifically, their approach does not allow for elimination of data structures threaded through repeated layers of monadic binding inside a recursive computation. Since the latter assertion seems somewhat in contradiction to the authors' description, let us elaborate on what we mean here.

First of all, the cases of successful fusion presented in [15,16] as examples all have the very specific form of a single consumer encountering a single producer, that is, eliminating exactly one layer of intermediate data structure. The authors suggest that sequences of $\gg\!=$ arising from **do**-notation lead to a rippling effect that enables several layers to be eliminated in a row, but we could not reproduce this. In particular, Ghani and Johann [16, Sect. 5] suggest that this happens for the following kind of monadic evaluator:

```
data Expr = Add Expr Expr | ...

eval (Add e1 e2) =
    do
      x ← eval e1
      y ← eval e2
      return (x+y)
...
```

But actually, the above right-hand side desugars to

$$\text{eval } e_1 \gg\!= (\lambda x \to \text{eval } e_2 \gg\!= (\lambda y \to \text{return } (x+y))) \tag{15}$$

rather than to an expression of the supposed form $(m \gg\!= k_1) \gg\!= k_2$. In fact, not a single invocation of the monadic short cut fusion rule is possible inside (15).

In contrast, our improvement approach is quite effective for eval. If, for example, we complete the above to

```
data Expr = ... | Div Expr Expr | Lit Int

...
eval (Div e1 e2) =
    do
      y ← eval e2
      if y = 0 then fail "division by zero" else
        do
          x ← eval e1
          return (div x y)
eval (Lit i) = return i
```

and run it against the exception monad defined as follows:

```
data F_Exc β = Fail String

instance Functor F_Exc where
    fmap h (Fail s) = Fail s

fail s = wrap (Fail s)
```

then we find that while improve does not necessarily always give asymptotic improvements, it still reduces absolute runtimes here. Moreover, it turns out to have a beneficial impact on memory requirements. In particular, for expressions with deeply nested computations, such as

```
deep n = foldl Add (Div (Lit 1) (Lit 0)) (map Lit [2..n])
```

we find that improve (eval (deep n)) :: Free F_Exc Int works fine for n that are orders of magnitude bigger than ones for which eval (deep n) :: Free F_Exc Int already leads to a stack overflow. An intuitive explanation here is that improve essentially transforms the computation into continuation-passing style.

Clearly, just as for eval above, the monadic short cut fusion method proposed by Ghani et al. [15,16] does not help with any of the earlier examples in this paper. Maybe it is possible to bring it to bear on such examples by inventing a suitable worker/wrapper scheme in the spirit of that applied by Gill [17] and Chitil [19] to achieve asymptotic improvements via short cut fusion. If that can be achieved at all for monadic short cut fusion, which is somewhat doubtful due to complications involving polymorphic recursion and higher-kinded types, it would definitely require extensive restructuring of the code to be improved, much in contrast to our near-transparent approach.

On the other hand, Ghani et al.'s work is ahead of ours in terms of the monads it can handle. Their fusion rule is presented for a notion of inductive monads that covers free monads as a special case. More specifically, free monads are inductive monads that arise as fixpoints of a bifunctor that, when applied to one argument, gives the functor sum of the constant-valued functor returning that fixed argument and some other arbitrary, but fixed, functor. In other words,

our Free ϕ corresponds to Mu (SumFunc ϕ) in the terminology of Ghani and Johann [16, Ex. 14]. Most fusion success stories they report are actually for this special kind of inductive monad and, as we have seen, all models of Swierstra and Altenkirch live in the free monad subspace as well. But still, it would be interesting to investigate a generalisation of our approach to inductive monads other than the free ones, in particular to ones based on functor product instead of functor sum above.

6 Conclusion

We have developed a program transformation that, in essence, makes monadic substitution a constant-time operation and can have further benefits regarding stack consumption. Using the abstraction mechanisms provided by Haskell's type system, we were able to formulate it in such a way that it does not interfere with normal program construction. In particular, programmers need not a priori decide to use the improved representation of free monads. Instead, they can program against the ordinary representation with the only (and non-encumbering) proviso that it be captured as one instance of an appropriate type constructor class. This gives code that is identically structured and equally efficient to the one they would write as usual. When utilisation of the improved representation is desired (for example, because a quadratic overhead is observed), dropping it in a posteriori is as simple as adding a single call to improve at the appropriate place. This transparent switching between the equivalent representations also means that any equational reasoning about the potentially to be improved code can be based on the ordinary representation, which is, of course, beneficial for applications like the ones of Swierstra and Altenkirch, discussed in Sect. 5.1. (Or, formulated in terms of the example from Sect. 2: we can reason and think about fullTree, as special case of fullTree′, even though actually fullTree″ will be run eventually.[1])

The genericity that comes via two-level types is a boon for developing and reasoning about our method, but not an indispensable ingredient. It is always possible to obtain type constructors, classes, and improve-functions tailored to a particular datatype (as in Sect. 2). This is done by bundling and unbundling type isomorphisms as demonstrated by Ghani and Johann [16, App. A].

Acknowledgements. I would like to thank the anonymous reviewers for their comments and suggestions.

[1] An example of a property that is much simpler to prove for fullTree than for fullTree″ is the fact that the output trees produced for input n will only ever contain integers from the interval 0 to n. While this has a straightforward proof by induction for fullTree n, proving it for fullTree″ n Leaf requires a nontrivial generalisation effort to find a good (i.e., general enough) induction hypothesis.

References

1. Moggi, E.: Notions of computation and monads. Information and Computation **93**(1) (1991) 55–92
2. Peyton Jones, S., Wadler, P.: Imperative functional programming. In: Principles of Programming Languages, Proceedings, ACM Press (1993) 71–84
3. Launchbury, J., Peyton Jones, S.: State in Haskell. Lisp and Symbolic Computation **8**(4) (1995) 293–341
4. Wadler, P.: The essence of functional programming (Invited talk). In: Principles of Programming Languages, Proceedings, ACM Press (1992) 1–14
5. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Principles of Programming Languages, Proceedings, ACM Press (1995) 333–343
6. Hughes, R.: A novel representation of lists and its application to the function "reverse". Information Processing Letters **22**(3) (1986) 141–144
7. Jones, M.: Functional programming with overloading and higher-order polymorphism. In: Advanced Functional Programming, Tutorial Text. Volume 925 of LNCS. Springer-Verlag (1995) 97–136
8. Sheard, T., Pasalic, E.: Two-level types and parameterized modules. Journal of Functional Programming **14**(5) (2004) 547–587
9. Reynolds, J.: Types, abstraction and parametric polymorphism. In: Information Processing, Proceedings, Elsevier (1983) 513–523
10. Wadler, P.: Theorems for free! In: Functional Programming Languages and Computer Architecture, Proceedings, ACM Press (1989) 347–359
11. Swierstra, W., Altenkirch, T.: Beauty in the beast — A functional semantics for the awkward squad. In: Haskell Workshop, Proceedings, ACM Press (2007) 25–36
12. Claessen, K., Hughes, R.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: International Conference on Functional Programming, Proceedings, ACM Press (2000) 268–279
13. Swierstra, W.: Data types à la carte. Journal of Functional Programming, to appear.
14. Swierstra, W., Altenkirch, T.: Dependent types for distributed arrays. In: Trends in Functional Programming, Draft Proceedings. (2008)
15. Ghani, N., Johann, P., Uustalu, T., Vene, V.: Monadic augment and generalised short cut fusion. In: International Conference on Functional Programming, Proceedings, ACM Press (2005) 294–305
16. Ghani, N., Johann, P.: Monadic augment and generalised short cut fusion. Journal of Functional Programming **17**(6) (2007) 731–776
17. Gill, A.: Cheap Deforestation for Non-strict Functional Languages. PhD thesis, University of Glasgow (1996)
18. Johann, P.: A generalization of short-cut fusion and its correctness proof. Higher-Order and Symbolic Computation **15**(4) (2002) 273–300
19. Chitil, O.: Type-Inference Based Deforestation of Functional Programs. PhD thesis, RWTH Aachen (2000)