Explicit *binds*: effortless efficiency with and without trees or *Love your enemy*

Tarmo Uustalu, Institute of Cybernetics
Joint work with Ralph Matthes, Université Paul Sabatier

Theory Days at Jõulumäe, 3–5 October 2008

This talk

How to make data-manipulating functions efficient? How to do so without obfuscating their definitions, keeping the natural definitions?

As a motivating example, we consider lists and the reverse function.

Two efficient representations: Lists with explicit appends and Church lists.

A generalization to free monads and explicit binds alt Church representation.

Further generalizations.

Standard lists

Recall lists in their natural form, as an inductive type with two constructors. Recall also fold.

```
data [e] = [] | e : [e]

foldL :: c -> (e -> c -> c) -> [e] -> c
foldL n c []       = n
foldL n c (e : es) = e `c` foldL n c es

(++) :: [e] -> [e] -> [e]
[]       ++ es' = es'
(e : es) ++ es' = e : (es ++ es')

-- EQUIVALENT DEFINITION VIA foldL
-- es ++ es' = foldL es' (:) es
```

Naive list reversal

```
sgltL :: e -> [e]
sgltL e = e : []

reverseL :: [e] -> [e]
reverseL [] = []
reverseL (e : es) = reverseL es ++ sgltL e

-- EQUIVALENT DEFINITION VIA foldL
-- reverseL = foldL [] (\ e esR -> esR ++ sgltL e)
```

This is the natural definition we would like to write. However, it is quadratic. . .

To append a value to a list, the whole list has to be traversed.

We need to append every element of the given list.

Reverse can be redefined using an accumulator, but this requires ad hoc work and means giving up the natural definition.

What happens?

```
> reverseL [0,1,2]

reverseL [1,2] ++ [0]
(reverseL [2] ++ [1]) ++ [0]
((reverseL [] ++ [2]) ++ [1]) ++ [0]

(([] ++ [2]) ++ [1]) ++ [0]          -- TRAVERSE []

([2] ++ [1]) ++ [0]                  -- TRAVERSE [2]
(2 : ([] ++ [1])) ++ [0]

[2,1] ++ [0]                         -- TRAVERSE [2,1]
2 : ([1] ++ [0])
2 : 1 : ([] ++ [0])

[2,1,0]
```

Other types of lists

We could conceive other types that have the same interface as the "naive" list type and implement the same functionality.

```
class List e es where
  nil :: es
  cons :: e -> es -> es
  fold :: c -> (e -> c -> c) -> es -> c
  app :: es -> es -> es

sglt :: List e es => e -> es
sglt e = e `cons` nil

reverse :: List e es => es -> es
reverse = fold nil (\ e esR -> esR `app` sglt e)

instance List e [e] where
  nil = []
  cons = (:)
  fold = foldL
  app = (++)
```

...used instead of standard lists

Such other representations can be used as reimplementations of the standard lists.

```
fromL :: List e es => [e] -> es
fromL es = foldL nil cons es

toL ::  List e es => es -> [e]
toL es = fold [] (:) es
```

Lists with explicit ("frozen") appends

Idea: treat folds of appends specifically by making append an additional constructor in the inductive type of lists and equipping fold with a fine-tuned additional clause for append.

```
data ListX e = Nil | e :< ListX e | ListX e :++ ListX e

instance List e (ListX e) where
  nil = Nil
  cons = (:<)

  fold n c Nil            = n
  fold n c (e :< es)      = e `c` fold n c es
  fold n c (es :++ es') = fold (fold n c es') c es          -- SMART

--fold n c (es :++ es') = fold n c (fold es' cons es)      -- NAIVE
--                        = fold n c (es `app` es')

  app = (:++)                                                -- SMART

--es `app` es' = fold es' cons es n                          -- NAIVE
```

Naive becomes smart!

The append function is may seem to be smart, but that's just the looks. She is entirely passive. The actual clever guy is the fold functional, who sponsors her.

Appends by themselves do nothing, things only happen when they are folded (eg at conversion from `ListX e` to `[e]`).

In particular, the naive reverse function has automagically become smart without effort: the naive definition

```
reverse :: List e es => es -> es
reverse = fold nil (\ e esR -> esR `app` sglt e)
```

is now linear!

```
> reverse (fromL [0,1,2,3])
((((Nil :++ (3 :< Nil)) :++ (2 :< Nil)) :++ (1 :< Nil)) :++ (0 :< Nil))

> toL (reverse (fromL [0,1,2,3]))
[3,2,1,0]
```

What happens?

```
> toL (((nil :++ sglt 2) :++ sglt 1) :++ sglt 0)

fold [] (:)
    (((nil :++ sglt 2) :++ sglt 1) :++ sglt 0)
fold (fold [] (:) (sglt 0)) (:)
    ((nil :++ sglt 2) :++ sglt 1)
fold (fold (fold [] (:) (sglt 0)) (:) (sglt 1)) (:)
    (nil :++ sglt 2)
fold (fold (fold (fold [] (:) (sglt 0)) (:) (sglt 1)) (:) (sglt 2)) (:)
    nil

fold (fold (fold [] (:) (sglt 0)) (:) (sglt 1)) (:) (sglt 2)
2 : fold (fold [] (:) (sglt 0)) (:) (sglt 1)
2 : 1 : fold [] (:) (sglt 0)
2 : 1 : 0 : []
```

Church lists

Idea (of Church representations): Identify lists by their fold functionals, so the general fold functional becomes just application. Define constructors as functions delivering adequate specialized fold functionals.

Idea: Treat append on par with constructors, ie specifically compared to other list functions, and choose its definition carefully.

```
data ListCh e = Build (forall x. x -> (e -> x -> x) -> x)

instance List e (ListCh e) where
  nil = Build (\ n c -> n)
  e ‘cons‘ Build f = Build (\ n c -> e ‘c‘ f n c)
--e ‘cons‘ es      = Build (\ n c -> e ‘c‘ fold n c es)

  fold n c (Build f) = f n c

  Build f ‘app‘ Build f’ = Build (\ n c -> f (f’ n c) c) -- SMART
--es      ‘app‘ es’      = Build (\ n c -> fold (fold n c es’) c es)

--Build f ‘app‘ es’ = f es’ cons                               -- NAIVE
```

Naive becomes smart again

Now the fold functional is just a dumb enabler for the constructors that really know what fold must do on any list.

The smartness is in the append function which knows how appends should be folded.

Again the naive reverse function becomes smart (linear rather than quadratic) just thanks to the smart append.

Let's compare and take stock

Lists with explicit appends: Smart clause in the definition of fold for a purely formal constructor of "frozen" append.

Church lists: Smart definition of append for a very plain fold.

The idea in both cases is exactly the same: to enforce a special treatment of folds of appends.

Both representations make it possible to implement this idea, but in different ways.

Append is a very crucial function in programming with lists. Mere handling of folds of appends efficiently can drastically optimize many list functions.

From list types to free monads

Lists and append are a special case of (wellfounded) leaf-labelled trees (with a fixed branching factor) and grafting.

The official name for these types is free monads. Grafting is the bind operation of such monads.

List types with explicit appends and Church lists generalize to free monads extended with with explicit bind ("frozen graft") operations and Church representations of free monads.

We get effortless efficiency for functions manipulating leaf-labelled trees.

Other generalizations

Functors and their fmap operations — any kind of labelled structures and relabelling

General "inductive monads" and their bind operations — like wellfounded leaf-labelled trees with grafting, but more liberal

Free completely iterative monads — non-wellfounded leaf-labelled trees with grafting and iteration

Cofree recursive comonads — wellfounded node-labelled trees with upwards accumulation and recursion

Nonempty list types — a special case; we get efficient causal dataflow computation (joint with Varmo Vene)

Lists with explicit maps: a glimpse

We use the inductive type

```
data ListX e = Nil | e :< ListX e | forall d . MapX (d -> e) (ListX d)
```

with an explicit map constructor.

We get an efficient version of prefixes naturally defined (for standard lists) as

```
prefixes :: [e] -> [[e]]
prefixes []       = []
prefixes (e : es) = [e] : map (e :) (prefixes es)
```

History of this all

Hughes — lists as functions for efficient reverse

Wadler — concatenate vanishes

Wadler, Gill — deforestation

Gill, Launchbury, Jones — shortcut deforestation (fold/build fusion)

Ghani, Uustalu, Vene — semantics of fold/build, augment for free monads and general "inductive monads"

Voigtländer — many sorts of clever tricks

Defunctionalization/refunctionalization

Kmett — ideas similar to this talk

Conclusion

Representations matter.

For specific kinds of types (eg functors, monads), consider taking special care of the specific operations (eg fmap, bind).

With luck, this alone can give big gains.

Forests as such are not a source of inefficiencies. Efficiency can be achieved both with inductive types and functions based representations.

One and the same idea of "positive discrimination" of potentially costly operations does the trick in both cases.