Functional Data Structures in Monoidal Categories

(work in progress)

Zhixuan Yang

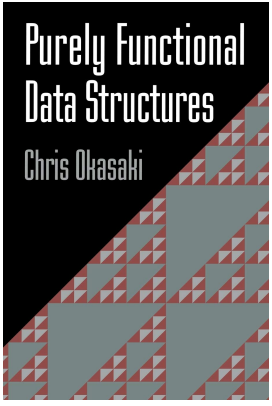
University of Exeter

Logic Colloquium 2026

Context

One of the greatest achievements of functional programming is the invention of many purely functional data structures.

Purity implies *persistence*, which is important in text editors, version control systems, databases, …



[Oka98]

Sequences

Sequences of data are ubiquitous in programming. A data structure for sequences is

$$
\begin{array}{ll}
S : \mathit{Type} \to \mathit{Type} & \mathit{nil} : S\ a \\
\mathit{cons} : a \to S\ a \to S\ a & \mathit{matchL} : b \to (a \to S\ a \to b) \to b \\
\mathit{snoc} : S\ a \to a \to S\ a & \mathit{matchR} : b \to (S\ a \to a \to b) \to b \\
(+\!\!+) : S\ a \to S\ a \to S\ a & \mathit{splitAt} : \mathit{Int} \to S\ a \to (S\ a, S\ a)
\end{array}
$$

satisfying certain equations.

Sequences

Sequences of data are ubiquitous in programming. A data structure for sequences is

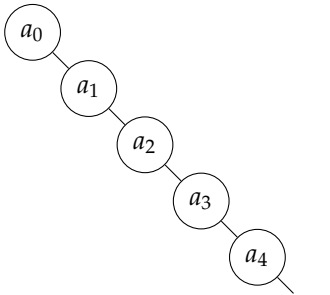
$$
\begin{array}{ll}
S : \mathit{Type} \to \mathit{Type} & \mathit{nil} : S\ a \\
\mathit{cons} : a \to S\ a \to S\ a & \mathit{matchL} : b \to (a \to S\ a \to b) \to b \\
\mathit{snoc} : S\ a \to a \to S\ a & \mathit{matchR} : b \to (S\ a \to a \to b) \to b \\
(+\!\!+) : S\ a \to S\ a \to S\ a & \mathit{splitAt} : \mathit{Int} \to S\ a \to (S\ a, S\ a)
\end{array}
$$

satisfying certain equations.

More operations (*head*, *tail*, *lookup*, *fold*, …) can be defined from the primitives above.

Cons lists

data *List* (*a* : *Type*) **where**
  Nil : *List a*
  Cons : *a* → *List a* → *List a*

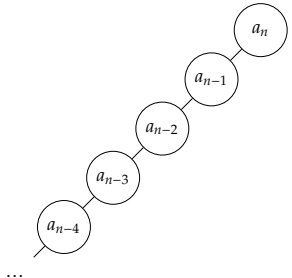


- Constant time for *cons* and *matchL*.
- Linear time for other operations.

Snoc lists

data *SnocList* (*a* : *Type*) **where**
  Nil : *SnocList a*
  Snoc : *SnocList a* → *a* → *SnocList a*

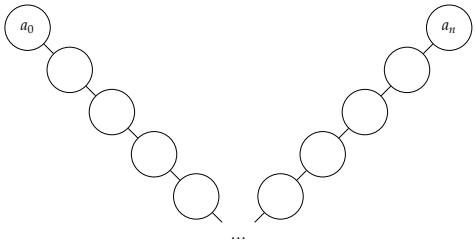
- Constant time for *snoc* and *matchR*.
- Linear time for other operations.



Batched Queue

data *BatchedQueue* (*a* : *Type*) **where**
Q : *List a* → *SnocList a* → *BatchedQueue a*

- Constant time for *cons* and *snoc*.
- Amortised constant time for *matchL* (in the partially persistent setting).

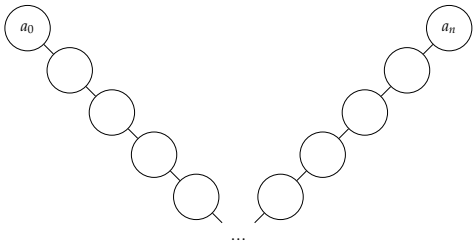


Banker Queue

data *BankerQueue* (*a* : *Type*) **where**
Q : *Int* → *List a* → *Int* → *SnocList a*
→ *BankerQueue a*

assuming laziness [Oka98].

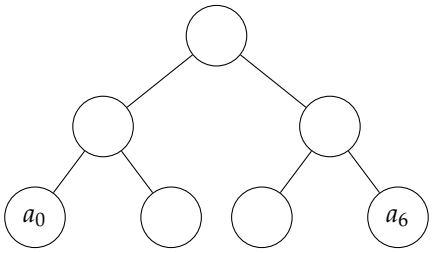
- Constant time for *cons* and *snoc*.
- Amortised constant time for *matchL* (in the fully persistent setting).



Length of the front half ⩾ length of the rear half.

Balanced Binary Trees

```
data AVLTrees (a : Type) where
  Nil : AVLTrees a
  Node : Int    -- tree height
       → AVLTrees a
       → a
       → AVLTrees a
       → AVLTrees
```



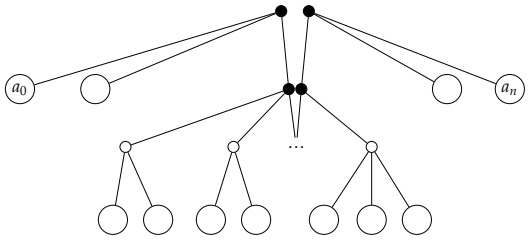
- Logarithmic time for all operations.

Finger Trees

data *FingerTrees* (*a* : *Type*) **where**

…

- Amortised constant time for *cons*, *snoc*, *matchL*, *matchR*.
- Logarithmic time for ⧺ and *splitAt*.

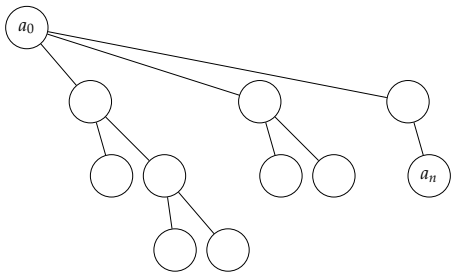


Catenable Lists

data *CList* (*a* : *Type*) **where**
Nil : *CList a*
Cons : *a* → *Queue* (*CList a*) → *CList a*

assuming laziness and some *Queue* supporting *snoc* and *matchL* in $O(1)$.

- Constant time for *cons*, *snoc*, ++, *matchL* [Oka98].
- Linear time for *matchR* and *splitAt*.



Essentially a rose tree with children of a node stored with a queue.

Main Idea

Observation

These data structures usually are implemented in languages with cartesian contexts (ML, Haskell, Agda, Idris, …), but *they use data linearly and orderly*.

Main Idea

Observation

These data structures usually are implemented in languages with cartesian contexts (ML, Haskell, Agda, Idris, …), but *they use data linearly and orderly*.

For example, concatenating two lists is

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

After all, why would a sequence structure discard/reorder/duplicate its data?

Main Idea

Idea

Linear type theories have semantics in suitable monoidal categories. Let's *interpret these data structures in suitable monoidal categories* to obtain new data structures!

Motivating Example

I am mostly interested in the monoidal category of *endofunctors and composition*.

$$
\left[\!\!\left[
\begin{array}{l}
\text{types } A \\
\text{terms } t \\
\text{linear product } A \otimes B \\
\text{lists } \mathit{List}\ A \cong 1 + A \otimes \mathit{List}\ A \\
\text{concatenation } (+\!\!+)
\end{array}
\right]\!\!\right]
\quad = \quad
\begin{array}{l}
\text{endofunctors } F \\
\text{natural transformation } \phi : F \dot{\rightarrow} G \\
\text{functor composition } F \circ G \\
\text{free monads } \mathit{Free}\ F \cong \mathit{Id} + F \circ \mathit{Free}\ F \\
\text{multiplication } \mu
\end{array}
$$

Motivating Example

I am mostly interested in the monoidal category of *endofunctors and composition*.

$$
\left[\!\!\left[
\begin{array}{l}
\text{types } A \\
\text{terms } t \\
\text{linear product } A \otimes B \\
\text{lists } \mathit{List}\ A \cong 1 + A \otimes \mathit{List}\ A \\
\text{concatenation } (+\!\!+)
\end{array}
\right]\!\!\right]
=
\begin{array}{l}
\text{endofunctors } F \\
\text{natural transformation } \phi : F \dot{\rightarrow} G \\
\text{functor composition } F \circ G \\
\text{free monads } \mathit{Free}\ F \cong \mathit{Id} + F \circ \mathit{Free}\ F \\
\text{multiplication } \mu
\end{array}
$$

Question

Can we obtain efficient implementations of free monads from efficient sequence data structures?

A Strange New World

Monoidal categories are a strange new world for doing data structures.

Strangeness 1

Lists and free monoids may differ when the monoidal category isn't closed.

To interpret the definition of $+\!\!+$, we need not only the universal property of the initial algebra $\mu X.\ I + a \otimes X$ but also the right adjoint $\mathit{List}\ a \multimap -$ to $- \otimes \mathit{List}\ a$.

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

A Strange New World

Example

Let $\langle \textsc{Mon}, \times, 1 \rangle$ be the cartesian monoidal category of monoids in $\textsc{Set}$. It is *not closed*.

- The free monoid over $A = \langle |A|, \cdot_A, \epsilon_A \rangle \in \textsc{Mon}$ is the free commutative monoid on A (following an Eckmann-Hilton argument).

A Strange New World

Example

Let $\langle \textsc{Mon}, \times, 1 \rangle$ be the cartesian monoidal category of monoids in $\textsc{Set}$. It is *not closed*.

- The free monoid over $A = \langle |A|, \cdot_A, \epsilon_A \rangle \in \textsc{Mon}$ is the free commutative monoid on A (following an Eckmann-Hilton argument).
- The list object $\mu X.\ 1 + A \times X$ over A has underlying set *List* $|A|$, unit [], and element-wise multiplication using $\cdot_A$ and ϵ_A.

Strangeness 2

The costs of accessing the two components of the monoidal product $A \otimes B$ may differ.

In the monoidal category of *Type*-endofunctors and composition,
$x : A, y : B \vdash (f\ x, y) : A' \otimes B$ is interpreted as $f_{BX} : A(BX) \to A'B(X)$ while
$x : A, y : B \vdash (x, g\ y) : A \otimes B'$ is interpreted as $A\ g_X : A(BX) \to A(B'X)$.

If A is a container functor, $A\ g_X$ needs to traverse over the container to apply g_X.

Strangeness 3

Isomorphic monoidal structures may have different cost characteristics.

Under suitable foundational assumptions, functor composition $A \circ B$ has the isomorphic co-Yoneda form $\int^{X:Type} AX \times (X \to B-)$, for which we do not need to map over A to access B.

A Strange New World

Strangeness 4

Snoc-lists and *Cons*-lists may differ, depending on how monoidal products $\otimes$ distributes over coproducts.

In the monoidal category of *Type*-endofunctors and composition, we have $(A + B) \circ C \cong (A \circ C) + (B \circ C)$ but not $A \circ (B + C) \circ (A \circ B) + (A \circ C)$.

Strangeness 4

Snoc-lists and *Cons*-lists may differ, depending on how monoidal products $\otimes$ distributes over coproducts.

In the monoidal category of *Type*-endofunctors and composition, we have $(A + B) \circ C \cong (A \circ C) + (B \circ C)$ but not $A \circ (B + C) \circ (A \circ B) + (A \circ C)$.

Consequently, letting $\mathit{Iter}\ A = \mu X.\ \mathit{Id} + (X \circ A)$,

$$\begin{aligned}
\mathit{Iter}\ A &\cong \mathit{Id} + \mathit{Iter}\ A \circ A \cong I + (I + \mathit{Iter}\ A \circ A) \circ A \\
&\cong \mathit{Id} + A + \mathit{Iter}\ A \circ A \circ A \\
&\cong \mathit{Id} + A + (A \circ A) + (A \circ A \circ A) + \cdots \\
&\ncong \mathit{Free}\ A
\end{aligned}$$

Strangeness 5

Case analysis of coproducts may only work under certain contexts, depending on how monoidal products $\otimes$ distributes over coproducts.

In an ordered linear context $(\Gamma_l, x : A + B, \Gamma_r)$, the semantics of pattern matching x needs distributivity of $\otimes$ over $+$, which we may not have.

Data Structures in Monoidal Categories

Cons lists and snoc lists work[1] in any monoidal categories with coproducts, relevant initial algebras, right adjoints $A \multimap -$ to $- \otimes A$ for all A.

[1] except *matchR* for cons lists, *cons*/$\dashv$ for snoc lists

Data Structures in Monoidal Categories

Cons lists and snoc lists work[1] in any monoidal categories with coproducts, relevant initial algebras, right adjoints $A \multimap -$ to $- \otimes A$ for all A.

It's a mis-conception that a snoc list is a reversed list. $\mathit{reverse} : \mathit{List}\ a \longrightarrow \mathit{List}\ a$ needs symmetry but *yank* does not.

$$
\begin{aligned}
&\mathit{yank} : \mathit{SnocList}\ a \longrightarrow \mathit{List}\ a \\
&\mathit{yank}\ sx = \mathit{go}\ xs\ \mathit{Nil}\ \textbf{where} \\
&\quad \mathit{go} : \mathit{SnocList}\ a \longrightarrow \mathit{List}\ a \longrightarrow \mathit{List}\ a \\
&\quad \mathit{go}\ \mathit{Nil}\ ys \qquad\quad = ys \\
&\quad \mathit{go}\ (\mathit{Snoc}\ sx\ x)\ ys = \mathit{go}\ sx\ (\mathit{Cons}\ x\ ys)
\end{aligned}
$$

[1] except *matchR* for cons lists, *cons*/$+\!\!+$ for snoc lists

Data Structures in Monoidal Categories

Banker queues work[2] in any monoidal category with *copowers*, coproducts, relevant initial algebras, right adjoints $A \multimap -$ to $- \otimes A$ for all A.

[2] *snoc*, *cons*, *matchL* works but not the others

Data Structures in Monoidal Categories

Banker queues work[2] in any monoidal category with *copowers*, coproducts, relevant initial algebras, right adjoints $A \multimap -$ to $- \otimes A$ for all A.

The size integers are used non-linearly. We should interpret them as copowers.

$$
\begin{aligned}
&\textbf{data}\ \mathit{BankerQueue}\ (a : \mathit{Type})\ \textbf{where}\\
&\quad Q : \mathit{Int} \to \mathit{List}\ a \to \mathit{Int} \to \mathit{SnocList}\ a \to \mathit{BankerQueue}\ a\\
&q : \mathit{Int} \to \mathit{List}\ a \to \mathit{Int} \to \mathit{SnocList}\ a \to \mathit{BankerQueue}\ a\\
&q\ \mathit{lenF}\ f\ \mathit{lenR}\ r = \textbf{if}\ \mathit{lenF} < \mathit{lenR}\\
&\quad \textbf{then}\ Q\ (\mathit{lenF} + \mathit{lenR})\ (f +\!\!+ \mathit{yank}\ r)\ 0\ \mathit{Nil}\\
&\quad \textbf{else}\ \ Q\ \mathit{lenF}\ f\ \mathit{lenR}\ r
\end{aligned}
$$

[2] *snoc*, *cons*, *matchL* works but not the others

Data Structures in Monoidal Categories

Balanced binary trees and finger trees work in any monoidal category with copowers, coproducts, relevant initial algebras, and *bi-closedness* (adjunctions $- \otimes A \dashv A \multimap -$ and $A \otimes - \dashv A \rightarrowtail -$ for all A). Symmetry is not necessary!

Data Structures in Monoidal Categories

Balanced binary trees and finger trees work in any monoidal category with copowers, coproducts, relevant initial algebras, and *bi-closedness* (adjunctions $- \otimes A \dashv A \multimap -$ and $A \otimes - \dashv A \rightarrowtail -$ for all A). Symmetry is not necessary!

Bi-closedness implies distributivity of $\otimes$ over $+$, allowing us to pattern match variables in any context:

$$
\begin{aligned}
&\mathit{rotateL}, \mathit{rotateR} : \mathit{AVLTree}\ a \longrightarrow \mathit{AVLTree}\ a \\
&\mathit{rotateL}\ (\mathit{Node}\ \_\ p\ x\ (\mathit{Node}\ \_\ q\ y\ r)) = \mathit{node}\ (\mathit{node}\ p\ x\ q)\ y\ r \\
&\mathit{rotateR}\ (\mathit{Node}\ \_\ (\mathit{Node}\ \_\ p\ x\ q)\ y\ r) = \mathit{node}\ p\ x\ (\mathit{node}\ q\ y\ r)
\end{aligned}
$$

With a small tweak, catenable lists work[3] in any monoidal category with copowers, coproducts, relevant initial algebras, $- \otimes A \dashv A \multimap -$. *No need for bi-closedness!*

$$
\begin{aligned}
&(+\!\!+) : \mathit{CList}\ a \longrightarrow \mathit{CList}\ a \longrightarrow \mathit{CList}\ a \\
&\quad \text{-- } \mathit{xs} +\!\!+ \mathit{Nil} = \mathit{xs} \\
&\mathit{Nil} +\!\!+ \mathit{ys} = \mathit{ys} \\
&C\ x\ \mathit{xss} +\!\!+ \mathit{ys} = C\ x\ (\mathit{snoc}\ \mathit{xss}\ \mathit{ys})
\end{aligned}
$$

We can't pattern match the second argument when xs is in the context. (This has complexity impact though).

[3] except for *matchR*/*splitAt*

Some Monoidal Categories

- *Symmetric closed*: cartesian closed ones, Day tensor products, tensor products of many algebraic structures, …

Some Monoidal Categories

- *Symmetric closed*: cartesian closed ones, Day tensor products, tensor products of many algebraic structures, …
- *Non-symmetric bi-closed*: endomorphism categories of the bi-categories of profunctors, relations, V-matrices, spans, …

Non-symmetric *by convention*

Some Monoidal Categories

- *Symmetric closed*: cartesian closed ones, Day tensor products, tensor products of many algebraic structures, …
- *Non-symmetric bi-closed*: endomorphism categories of the bi-categories of profunctors, relations, V-matrices, spans, …

Non-symmetric *by convention*

- *Non-symmetric one-sided-closed*: endofunctor and composition, lexicographic products, semi-direct products [Ful16]

Non-symmetric *by nature*

Type Formers à la Carte

We don't want a linear language with a fixed set of type formers, but a menu for *type formers à la carte*:

- Ordered linear contexts
- Exchangeable linear contexts
- Linear products/functions
- Cartesian products/functions
- Coproducts
- Fixed-point types
- Linear/non-linear adjunctions
- …

Typing rules for these are standard [Ben95, PP99, Lam58, YW26, JM10].

Type Formers à la Carte

I am working on a Haskell library `linearity-lab` for linear type formers à la carte.

- all aforementioned type formers and the semantics in *Functor* are implemented;
- a programming library, *not a formalisation* – general recursion, recursive types, no equational laws.
- only ordered contexts and zero-or-one-variable contexts (as in CBPV) for now, but exchangeable contexts and graded contexts are planned for future.

Implementation techniques are heavily insipired by [PZ17].

Implementation Design Choices

Design Choice

Type formers are formulated as Haskell type-classes.

For example, the two rules for ordered linear contexts (variable and substitution)

$$\frac{}{x : t \vdash x : t} \qquad \frac{\Delta \vdash M : t_1 \qquad \Gamma_l, x : t_1, \Gamma_r \vdash N : t_2}{\Gamma_l, \Delta, \Gamma_r \vdash \mathit{let}\ x = M\ \mathit{in}\ N : t_2}$$

becomes

$$\begin{aligned}
&\textbf{class}\ \mathit{HasOrdCtx}\ (\mathit{tm} :: \mathit{LCtx} \to \mathit{LType} \to \mathit{Type})\ \textbf{where} \\
&\quad \mathit{var} :: \mathit{tm}\ [(x, t)]\ t \\
&\quad \mathit{llet_} :: \mathit{tm}\ \mathit{ctx}\ t_1 \to \mathit{tm}\ (\mathit{ctxL} +\!\!+ [(x, t_1)] +\!\!+ \mathit{ctxR})\ t_2 \\
&\qquad \to \mathit{tm}\ (\mathit{ctxL} +\!\!+ \mathit{ctx} +\!\!+ \mathit{ctxR})\ t_2
\end{aligned}$$

Implementation Design Choices

Design Choice

Use *numerically named variables and equality constraints* for better type inference.

With named variables, we can compute the left and right contexts $ctxL, ctxR$ from the middle context ctx and the whole context ctx'. We define

$$
\begin{aligned}
&\mathit{llet} :: (\mathit{HasCtx}\ tm, ctx' \sim ctxL +\!\!+ ctx +\!\!+ ctxR, (ctxL, ctxR) \sim \mathit{Split}\ ctx\ ctx') \\
&\quad \Rightarrow tm\ ctx\ t_1 \\
&\quad \rightarrow (tm\ [(\mathit{Fresh}\ ctx', t_1)]\ t_1 \rightarrow tm\ (ctxL +\!\!+ [(\mathit{Fresh}\ ctx', t_1)] +\!\!+ ctxR)\ t_2) \\
&\quad \rightarrow tm\ ctx'\ t_2 \\
&\mathit{llet}\ x\ body = \mathit{llet\_}\ @tm\ @ctxL\ @ctx\ @ctxR\ n\ x\ (body\ var)
\end{aligned}
$$

Implementation Design Choices

Design Choice

Use *numerically named variables and equality constraints* for better type inference.

Consequently supposing $u :: tm\ [(1, B)]\ D$, we don't have to write

$$t :: tm\ [(0, A), (1, B), (2, C)]\ E$$
$$t = llet\_\ @[(0, A)]\ @[(2, C)]\ u\ (\lambda b \to \ldots)$$

because *llet* can compute $Split\ [(1, B)]\ [(0, A), (1, B), (2, C)] = ([(0, A)], [(2, C)])$

$$t :: tm\ [(0, A), (1, B), (2, C)]\ E$$
$$t = llet\ u\ (\lambda b \to \ldots)$$

Without uniquely named variables this can't be done.

Implementation Design Choices

Design Choice

Variable terms are passed around at the meta level.

We can write terms like $llet\ u\ (\lambda b \longrightarrow b)$ because

$$
\begin{aligned}
&llet :: (\ldots) \Rightarrow tm\ ctx\ t_1 \\
&\quad \longrightarrow (tm\ [(Fresh\ ctx', t_1)]\ t_1 \longrightarrow tm\ (ctxL +\!\!+ [(Fresh\ ctx', t_1)] +\!\!+ ctxR)\ t_2) \\
&\quad \longrightarrow tm\ ctx'\ t_2
\end{aligned}
$$

It feels like programming with HOAS, but under the hood there is no magic:

$$llet\_ :: tm\ ctx\ t_1 \longrightarrow tm\ (ctxL +\!\!+ [(x, t_1)] +\!\!+ ctxR)\ t_2 \longrightarrow tm\ (ctxL +\!\!+ ctx +\!\!+ ctxR)\ t_2$$

Some Example Programs

Define $L\ a$ as the fixed-point $L\ a \cong One + (a \otimes L\ a)$:

$$\textbf{type}\ L :: LType \longrightarrow LType \qquad \textbf{data}\ L\_\ (a :: LType)$$
$$\textbf{type}\ L\ (a :: LType) = LFix\ (L\_\ a) \qquad \textbf{type instance}\ Decode\ (L\_\ a)\ x = One + (a \otimes x)$$

Some Example Programs

Define $L\ a$ as the fixed-point $L\ a \cong One + (a \otimes L\ a)$:

$$\textbf{type}\ L :: LType \longrightarrow LType \qquad \textbf{data}\ L\_\ (a :: LType)$$
$$\textbf{type}\ L\ (a :: LType) = LFix\ (L\_\ a) \qquad \textbf{type instance}\ Decode\ (L\_\ a)\ x = One + (a \otimes x)$$

The type formers we need for lists:

$$\textbf{type}\ HasList\ tm = (HasOrdCtx\ tm, HasLFun\ tm, HasLFix\ tm, HasLProd\ tm, HasCoProd\ tm)$$

Some Example Programs

Define $L\ a$ as the fixed-point $L\ a \cong One + (a \otimes L\ a)$:

$$\textbf{type}\ L :: LType \to LType \qquad \textbf{data}\ L\_\ (a :: LType)$$
$$\textbf{type}\ L\ (a :: LType) = LFix\ (L\_\ a) \qquad \textbf{type instance}\ Decode\ (L\_\ a)\ x = One + (a \otimes x)$$

The type formers we need for lists:

$$\textbf{type}\ HasList\ tm = (HasOrdCtx\ tm, HasLFun\ tm, HasLFix\ tm, HasLProd\ tm, HasCoProd\ tm)$$

Programming in the order linear language isn't too different from ordinary Haskell:

$$cayley :: (HasList\ tm, \ldots) \Rightarrow tm\ ctx\ (L\ a) \to tm\ ctx\ (L\ a \multimap L\ a)$$
$$cayley\ xs = matchL\ xs\ (abs\ \$\ \lambda x \to x)\ (\lambda x\ xs \to abs\ (\lambda ys \to consL\ x\ (cayley\ xs\ \hat{}\ ys)))$$
$$app :: (HasList\ tm, \ldots) \Rightarrow tm\ ctx_1\ (L\ a) \to tm\ ctx_2\ (L\ a) \to tm\ ctx\ (L\ a)$$
$$app\ xs\ ys = cayley\ xs\ \hat{}\ ys$$

Some Example Programs

We obtain free monads by interpreting lists in the endofunctor category:

$$\begin{aligned}
&\textbf{type}\ \mathit{Free} :: (\mathit{Type} \to \mathit{Type}) \to (\mathit{Type} \to \mathit{Type}) \\
&\textbf{type}\ \mathit{Free}\ f = \mathsf{Fct}[\![(L\ (\mathit{Fct}\ f))]\!] \\
&\mathit{injFree} :: \forall f\ x.\ \mathit{Functor}\ f \Rightarrow f\ x \to \mathit{Free}\ f\ x \\
&\mathit{injFree} = \mathit{asFctMap}\ @[\mathit{Fct}\ f]\ \$\ \lambda(\mathit{Vars}\ c) \to \mathit{singL}\ c \\
&\mathit{joinFree} :: \forall f\ x.\ \mathit{Functor}\ f \Rightarrow \mathit{Free}\ f\ (\mathit{Free}\ f\ x) \to \mathit{Free}\ f\ x \\
&\mathit{joinFree} = \mathit{asFctMap}\ @[L\ (\mathit{Fct}\ f), L\ (\mathit{Fct}\ f)]\ \$\ \lambda(\mathit{Vars}\ (c, k)) \to \mathit{appL}\ c\ k
\end{aligned}$$

Along this line, we can interpret *CList* to obtain an efficient implementation of free monads (not shown here for the lack of time).

Compare with a Manual Translation

Without going through the abstraction of linear logics and monoidal categories, I don't think I can come up with *CListF* directly:

```
data Free f a where
  Ret :: a → Free f a
  Cons :: f a → (a → Free f b) → Free f b
```

```
data SnocF f a where
  Ret' :: a → SnocF f a
  Snoc :: SnocF f a → (a → f b) → SnocF f b
```

```
data QueueF f a where
  Q :: Int → Free f a → Int → (a → SnocF f b) → QueueF f b
```

```
data CListF f a where
  E :: a → CListF f a
  C :: f b → (b → QueueF (CListF f) a) → CListF f a
```

Application

CListF is useful for abstract syntax with efficient substitution ($O(1)$-per-variable).

- Substitution of free monads is *not capture-avoiding*. If we want capture-avoiding, we can work in the category of nominal sets.
- An efficient implementation of name abstraction of nominal sets $[\mathbb{A}]X$ is via stateful fresh-name generation.

Note however, in some situations efficient substitution can be implemented more easily, such as having an 'environemnt' [Yan26].

Conclusion

Things I didn't talk about today:

- a careful complexity analysis of *CListF*
- the need of memoization for $(f \circ g)\ y = \exists x.\, f\ x \times (x \longrightarrow g\ y)$
- embedding functors in profunctors for performance

Future directions:

- Stateful algorithms in moniodal categories
- Laziness in moniodal categories
- Algorithmic complexity in moniodal categories

References I

- P. N. Benton, *A mixed linear and non-linear logic: Proofs, terms and models: Extended abstract*, p. 121–135, Springer Berlin Heidelberg, 1995.
- Ben Fuller, *Semidirect products of monoidal categories*, 2016.
- Mauro Jaskelioff and Eugenio Moggi, *Monad transformers as monoid transformers*, Theoretical Computer Science **411** (2010), no. 51-52, 4441–4466.
- Joachim Lambek, *The mathematics of sentence structure*, The American Mathematical Monthly **65** (1958), no. 3, 154–170.
- Chris Okasaki, *Purely functional data structures*, Cambridge University Press, 1998.

References II

- Jeff Polakow and Frank Pfenning, *Relating natural deduction and sequent calculus for intuitionistic non-commutative linear logic*, Electronic Notes in Theoretical Computer Science **20** (1999), 449–466.
- Jennifer Paykin and Steve Zdancewic, *The linearity monad*, Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, ICFP '17, ACM, 2017, p. 117–132.
- Zhixuan Yang, *An algorithmic reconstruction of normalisation by evaluation*, 2026.
- Zhixuan Yang and Nicolas Wu, *Modular models of monoids with operations by lifting functors along fibrations*, Journal of Functional Programming **Volume 36** (2026).