Partially Static Data as Free Extension of Algebras

Jeremy Yallop

and

Tamara von Glehn

and

Ohad Kammar

Partially-static data

Background & motivation

Multi-stage programming: a short introduction

Multi-stage programming
is about
control over β-**reduction**

$[\![\mathsf{e}]\!]$

do not reduce e

$\$\mathsf{e}$

(inside $[\![\ldots]\!]$)
reduce e

Multi-stage programming, motivated

power in one stage:

```
power :: Int → Int → Int
power x 0 = 1
power x n = x * power x (n - 1)
```

```
λ> power 2 6
64
```

Multi-stage programming by example

`power` in multiple stages (first exponent, then base):

```
power :: Code Int → Int → Code Int
power x 0 = ⟦1⟧
power x n = ⟦$x * $(power ⟦x⟧ (n - 1))⟧
```

```
λ> ⟦\x → $(power ⟦x⟧ 6) ⟧
⟦\x → x * (x * (x * (x * (x * (x * 1))))) ⟧
```

Partially-static data, motivated

With **control over** β only, generated code is inefficient:

```
λ> ⟦\x → $(power ⟦x⟧ 6) ⟧
⟦\x → x * (x * (x * (x * (x * (x * 1))))) ⟧
```

With support for **algebraic laws** we can generate better code:

```
⟦\x → let y = x * x in let z = y * y in z * y ⟧
```

Partially-static data

Building *equation-aware* structures

Plan: *drop-in* replacements for

$\langle$`String`,`++`$\rangle$

$\langle$`Int`,`+`,`*`$\rangle$

$\langle$`Bool`, $\wedge$, $\vee\rangle$

etc.!

Magma, a minimal structure

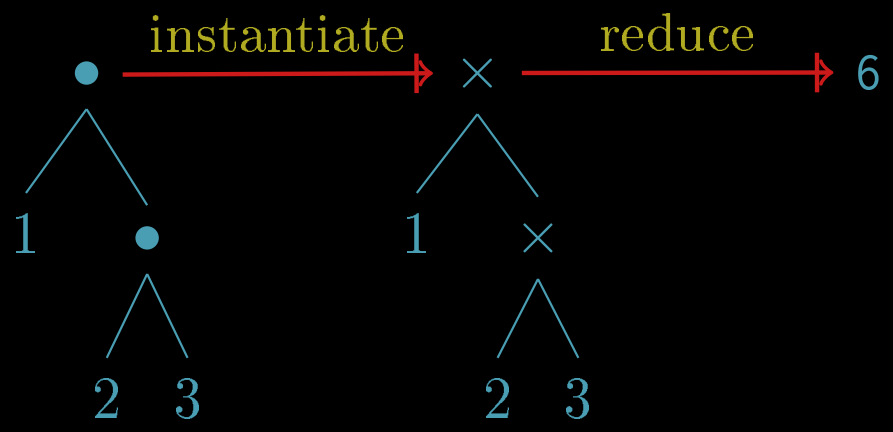
```
class Magma a where (•) :: a → a → a
```

Instances of Magma

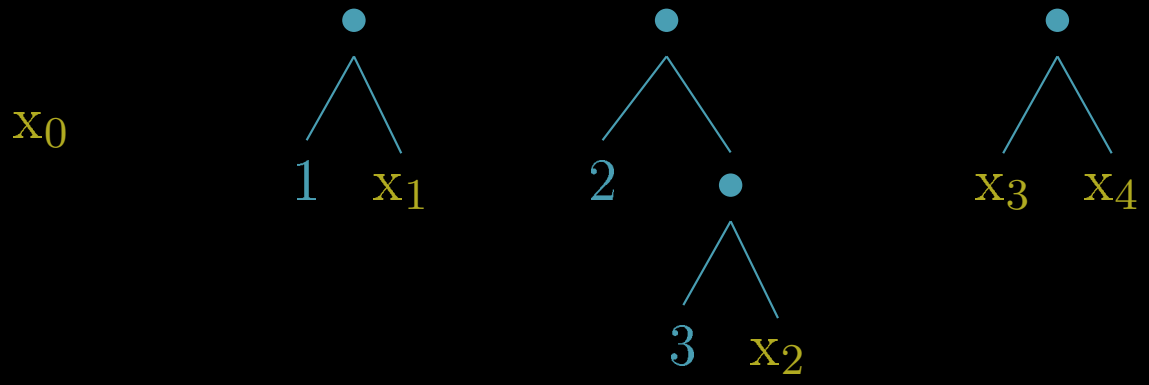
```haskell
newtype Int× = Int× Int
```

```haskell
instance Magma Int× where
  Int× x • Int× y = Int× (x × y)
```

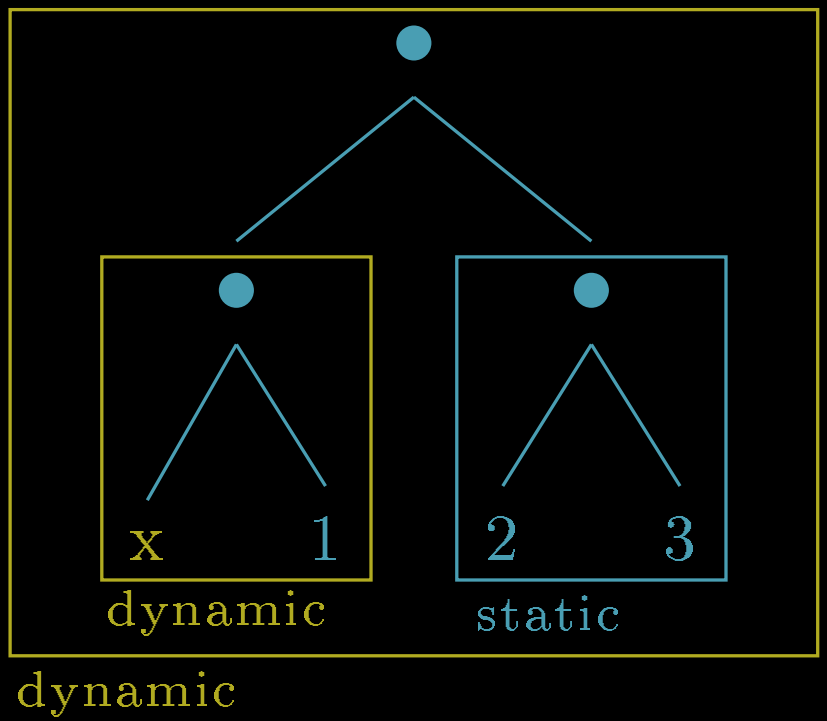
Reducing terms



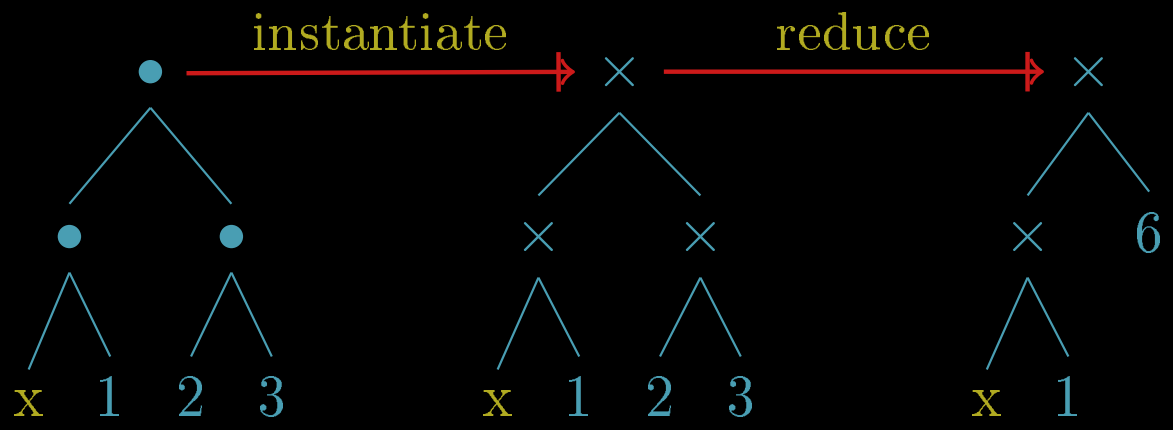
Trees with free variables



Binding-time analysis



Reducing terms with free variables



Back to Haskell: binding times

```haskell
data BindingTime =
     Sta -- available now
   | Dyn -- available later
```

```haskell
data BT :: BindingTime → * where
  BTSta :: BT Sta
  BTDyn :: BT Dyn
```

Possibly-static data (for leaves)

```
data SD :: BindingTime → * → * where
  S ::      a → SD Sta a
  D :: Code a → SD Dyn a
```

```
btSD :: SD bt a → BT bt
btSD (S _) = BTSta
btSD (D _) = BTDyn
```

Mixed magmas: binding-time-indexed normal forms

```
data Mag :: BindingTime → * → * where
  LeafM :: SD bt a → Mag bt a
  Br1 :: Mag Sta a → Mag Dyn a → Mag Dyn a
  Br2 :: Mag Dyn a → Mag r   a → Mag Dyn a
```

```
btMag :: Mag bt a → BT bt
btMag (LeafM m) = btSD m
btMag (Br1 _ _) = BTDyn
btMag (Br2 _ _) = BTDyn
```

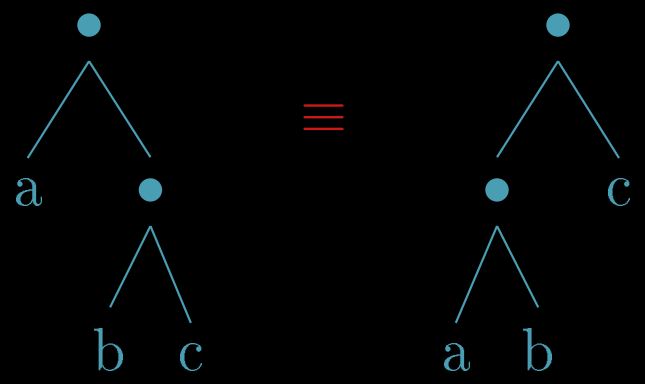
Mag is a Magma

```
instance Magma a ⇒ Magma (Exists Mag a) where
 E a • E b = m (btMag a) (btMag b) a b
 where
  -- leave no static subtrees!
  m BTSta BTSta (LeafM (S a)) (LeafM (S b)) = E (LeafM (S (a • b)))
  m BTSta BTDyn      l              r       = E (Br1 l r)
  m BTDyn   _        l              r       = E (Br2 l r)
```

A general-purpose existential type:

```
data Exists :: (k₁ → k₂ → *) → k₂ → * where
   E :: f b a → Exists f a
```

Semigroups (magmas + associativity)

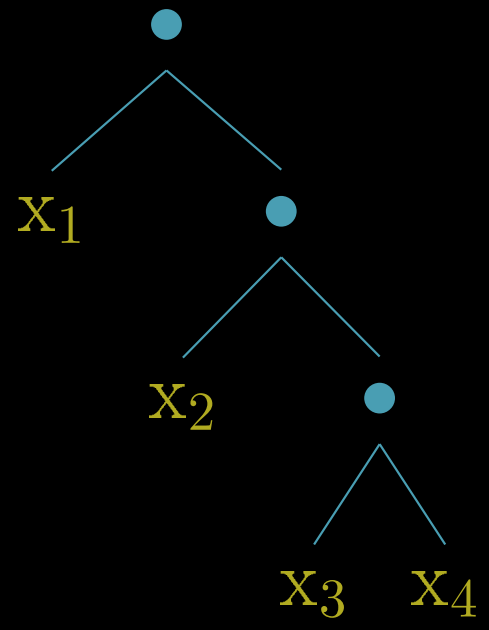



```
class Magma a ⇒ Semigroup a    -- a • (b • c) ≡ (a • b) • c
```

```
instance Semigroup Int×
```

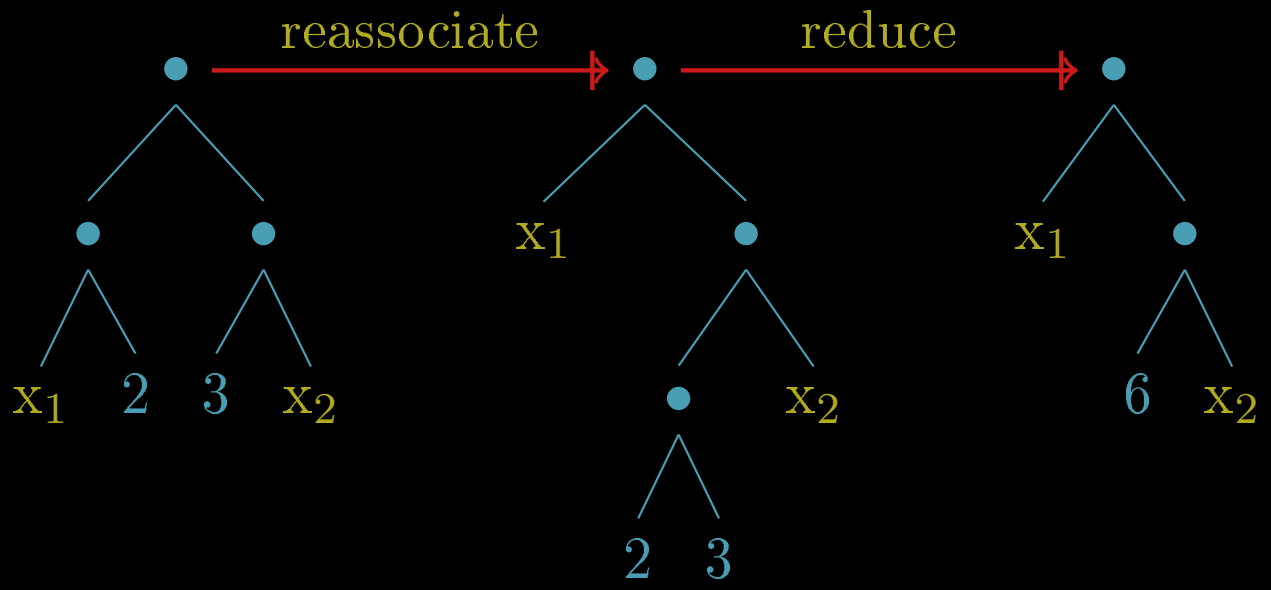
Normal forms for semigroups

Plain semigroups: fully right-associated



Mixed semigroups: also, no adjacent static data

Normalizing mixed-stage semigroup trees



Mixed semigroups: binding-time-indexed normal forms

```
data Semi :: BindingTime → * → * where
  LeafS :: SD bt a → Semi bt a
  ConsS ::       a → Semi Dyn a → Semi Dyn a
  ConsD ::  Code a → Semi r   a → Semi Dyn a
```

cons a static element:

```
consS :: Magma a ⇒ a → Exists Semi a → Exists Semi a
consS h (E (LeafS (S s)))   = E (LeafS (S (h • s)))
consS h (E t@(LeafS (D _))) = E (ConsS h t)
consS h (E (ConsS s t))     = E (ConsS (h • s) t)
consS h (E t@(ConsD _ _))   = E (ConsS h t)
```

cons a dynamic element:

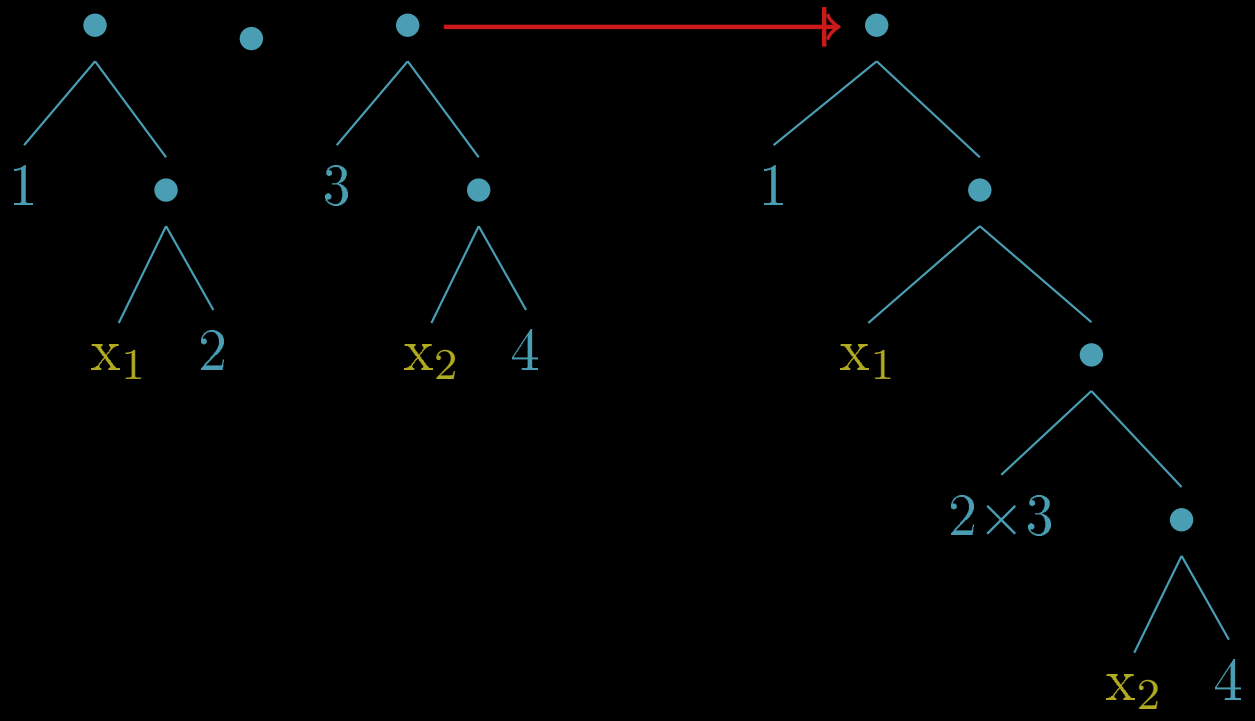
```
consD :: Code a → Exists Semi a → Exists Semi a
consD h (E t) = E (ConsD h t)
```

Semi is a Semigroup

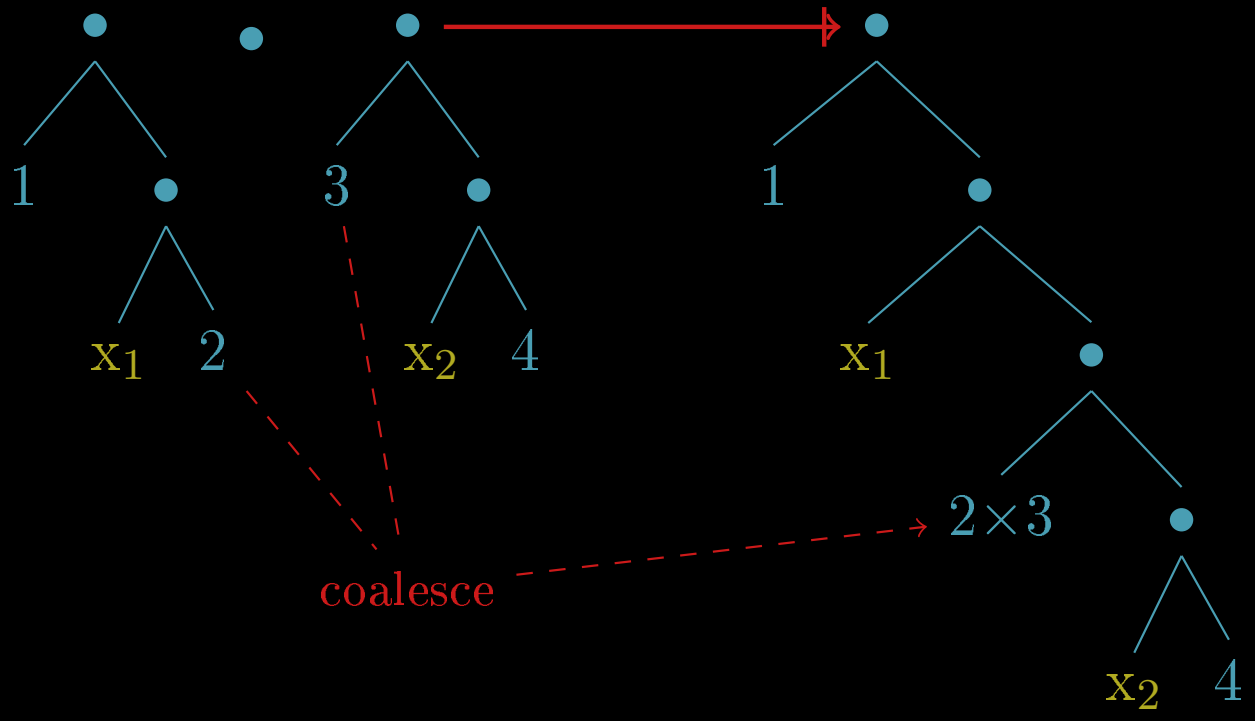
```haskell
instance Semigroup a ⇒ Magma (Exists Semi a)
 -- • traverses the entire left operand
 where E (LeafS (S s)) • l = consS s l
       E (LeafS (D d)) • l = consD d l
       E (ConsS h t) • l = consS h (E t • l)
       E (ConsD h t) • l = consD h (E t • l)
```

```haskell
instance Semigroup a ⇒ Semigroup (Exists Semi a)
```

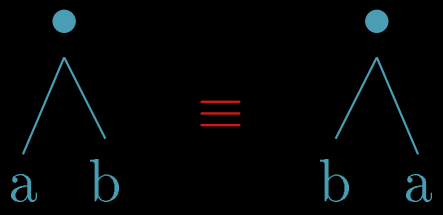
- maps normal forms to normal forms



- maps normal forms to normal forms

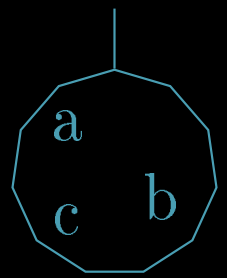


Adding commutativity



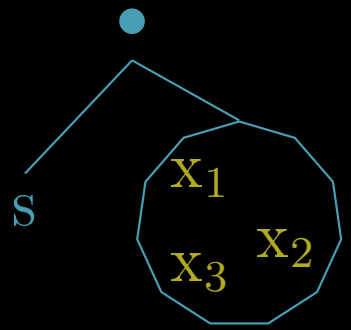

```
class Semigroup a ⇒ CSemigroup a    -- a • b ≡ b • a
```

A new n-ary constructor: unordered children



Partially-static commutative semigroups: normal forms

Group together all static data & all dynamic data:




```
data CSemi a = CSemi (Maybe a) (MultiSet (Code a))
```

CSemi is a CSemigroup

```
instance CSemigroup a ⇒ Magma (CSemi a) where
  CSemi s₁ d₁ • CSemi s₂ d₂ = CSemi (s₁ •? s₂) (union d₁ d₂)
   where Nothing •? m = m
         m •? Nothing = m
         Just m •? Just n = Just (m • n)

instance CSemigroup a ⇒ Semigroup (CSemi a)
instance CSemigroup a ⇒ CSemigroup (CSemi a)
```

Partially-static data

General structure

Requirements (rough sketch)

```
-- type of partially-static data
--   (parameterised by class)
PS :: (* → Constraint) → * → *

-- injection of static values
sta :: algebra a ⇒ a → PS algebra a

-- injection of dynamic values
dyn :: Code a → PS algebra a

-- turn partially-static values into dynamic
cd :: PS algebra a → Code a
```

Example: sta and dyn for CSemigroup

$$\mathsf{sta}_{\mathsf{CS}} = \lambda \mathsf{s} \to \mathsf{CSemi\ (Just\ s)\ empty}$$

$$\mathsf{dyn}_{\mathsf{CS}} = \lambda \mathsf{d} \to \mathsf{CSemi\ Nothing\ (singleton\ d)}$$

Question: How should we define the general PS?

Ingredient 1: coproducts

```
class (algebra a, algebra b, algebra (Coprod algebra a b)) ⇒
 Coproduct algebra a b
  where
   -- coproduct representation (varies with algebra)
   data family Coprod algebra a b :: *

   -- injections
   inl :: a → Coprod algebra a b
   inr :: b → Coprod algebra a b

   -- eliminator/fold
   eva :: algebra c ⇒
      (a → c) → (b → c) → Coprod algebra a b → c
```

eva f g (inl s_1 • inr d_1 • ...) $\rightsquigarrow$ f s_1 • g d_1 • ...

Ingredient 2: free objects

```
class algebra (FreeA algebra x) ⇒ Free algebra x where
  -- free object representation (varies with algebra)
  data family FreeA algebra x :: *

  -- variable injection
  pvar :: x → FreeA algebra x

  -- eliminator/fold
  pbind :: algebra c ⇒ FreeA algebra x → (x → c) → c
```

```
pbind f (pvar x₁ • pvar x₂ • ... )   ⇝   f x₁ • f x₂ • ...
```

Free extensions from coproducts & free objects

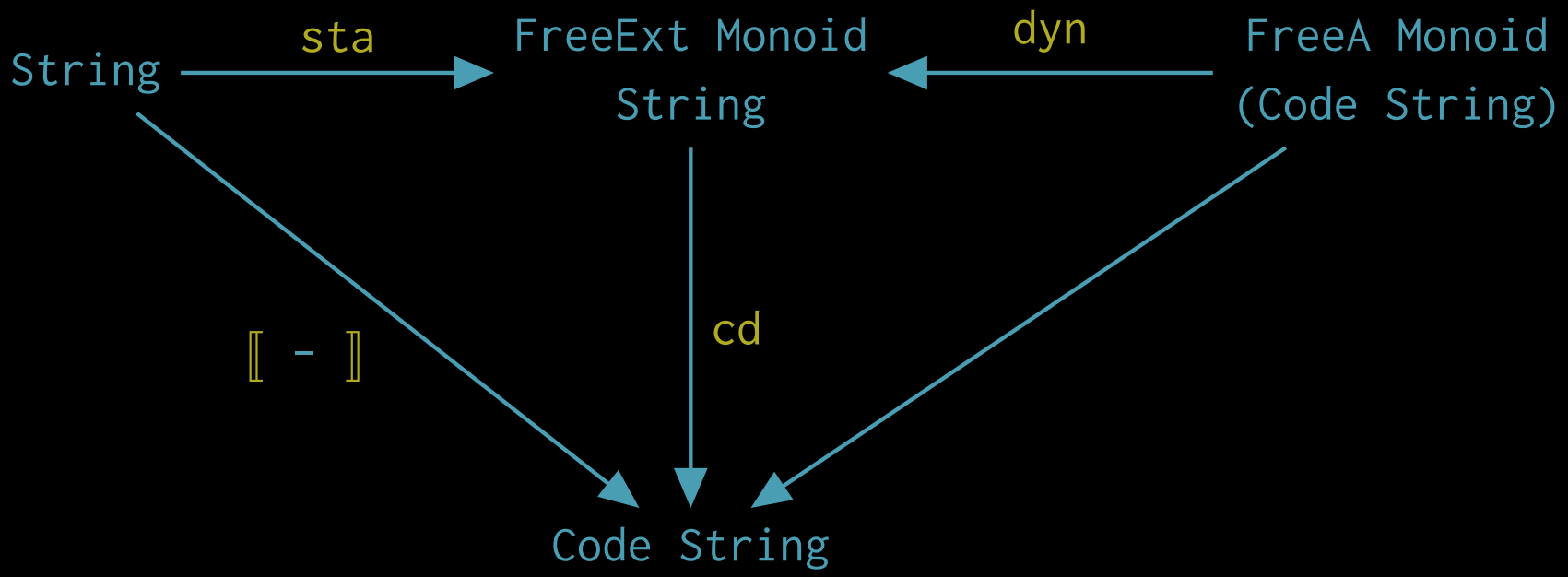
Free extension constraints:

```
FreeExt_C :: (* → Constraint) → * → Constraint

type FreeExt_C algebra a =
   Coproduct algebra a (FreeA algebra (Code a))
```

Free extension types:

```
FreeExt :: (* → Constraint) → * → *

type FreeExt algebra a =
   Coprod algebra a (FreeA algebra (Code a))
```

Free extensions, pictured



frex interface: sta and dyn

```
-- sta: left injection into the free extension
sta :: (algebra a, FreeExt_C algebra a) ⇒
       a → FreeExt algebra a
sta = inl
```

```
-- dyn: right injection of variables into the free extension
dyn :: (Free algebra (Code a), FreeExt_C algebra a) ⇒
       Code a → FreeExt algebra a
dyn = inr · pvar
```

frex interface: cd

```
-- cd: elimination of free extensions into code
cd :: (Lift a, Free algebra (Code a), algebra (Code a),
       FreeExt_C algebra a) ⇒
      FreeExt algebra a → Code a
cd = eva tlift (`pbind` id)
```

```
-- (tlift turns static values into code)
tlift :: Lift a ⇒ a → Code a
tlift = liftM TExp · lift
```

Partially-static data

Instances & applications

Algebras and their free extensions

Algebra	**Free extension**
monoids	alternating static/dynamic sequence
commutative monoids	(static element) $\times$ (bag of names)
commutative rings	multinomial
distributive lattices	multinomial (exponents 0 or 1)
F-algebras	free algebra of coproducts
sets	binary sum

Coproduct of monoids

```
class Monoid t where
  𝟙 :: t
  (⊛) : t → t → t
```

The coproduct is an alternating sequence:

```
data AorB = A | B
data Alternate :: AorB → * → * → * where
   Empty :: Alternate any a b
   ConsA :: a → Alternate B a b → Alternate A a b
   ConsB :: b → Alternate A a b → Alternate B a b
```

```
instance (Monoid a, Monoid b) ⇒ Coproduct Monoid a b where
   data Coprod Monoid a b where M :: Alt _ a b → Coprod Monoid a
   inl a = M (ConsA a Empty)
   inr b = M (ConsB b Empty)
   ...
```

The free monoid

```
instance Free Monoid x where
  newtype FreeA Monoid x = P [x] deriving (Monoid)
  pvar x = P [x]
  P [] `pbind` f = 𝟙
  P xs `pbind` f = foldr ((⊛) · f) 𝟙 xs
```

Using the monoid free extension

`printf`:

```
sprintf ((int ⊕ lit "a") ⊕ (lit "b" ⊕ int))
```

`printf`, staged with ⟦ ⟧ and $:

```
⟦ λx y →(((("" ++ show x) ++ "a") ++ "b") ++ show y ⟧
```

`printf`, staged with partially-static data:

```
⟦ λx y → show x ++ "ab" ++ show y ⟧
```

Using the monoid free extension

`printf`:

```
sprintf ((int ⊕ lit "a") ⊕ (lit "b" ⊕ int))
```

`printf`, staged with ⟦ ⟧ and $:

```
⟦ λx y →(((("" ++ show x) ++ "a") ++ "b") ++ show y ⟧
```

`printf`, staged with partially-static data:

```
⟦ λx y → show x ++ "ab" ++ show y ⟧
```

Using the monoid free extension

`printf`:

```
sprintf ((int ⊕ lit "a") ⊕ (lit "b" ⊕ int))
```

`printf`, staged with ⟦ ⟧ and $:

```
⟦ λx y →(((("" ++ show x) ++ "a") ++ "b") ++ show y ⟧
```

`printf`, staged with partially-static data:

```
⟦ λx y → show x ++ "ab" ++ show y ⟧
```

Free extension of commutative monoids

```
class Monoid m ⇒ CMonoid m
```

The coproduct is a product!

```
instance (CMonoid a, CMonoid b) ⇒ Coproduct CMonoid a b where
  data Coprod CMonoid a b = C a b
  inl a = C a 𝟙
  inr b = C 𝟙 b
  eva f g (C a b) = f a ⊛ g b
```

The free object is a bag

```
instance Ord x ⇒ Free CMonoid x where
  newtype FreeA CMonoid x = CM (MultiSet x)
  ...
```

Using the commutative monoid free extension

`power`

```
power 5 ⟦x⟧
```

`power`, staged with ⟦ ⟧ and $:

```
⟦ 1 * (x * (x * (x * (x * x)))) ⟧
```

`power`, with partially-static data:

```
⟦ let y = x * x in let z = y * y in x * z ⟧
```

Free commutative rings

```
class Ring a where
  (⊕), (⊗) :: a → a → a
  rneg :: a → a
  𝕆, 𝟙 :: a
```

Free rings are multinomials with integer coefficients:

```
data Multinomial x a = MN (Map (MultiSet x) a)

instance Ord x ⇒ Free Ring x where
  newtype FreeA Ring x = RingA (Multinomial x Int)
  pvar x = RingA (MN (singleton (singleton x) 𝟙))
  RingA xss `pbind` f = evalMN initMN f xss
```

Free extension of commutative rings

No closed form for coproducts. But can define free extension!

Free extension: multinomials with coefficients in `a`:

```
instance (Ring a,Ord x) ⇒ Coproduct Ring a (FreeA Ring x) where
  newtype Coprod Ring a (FreeA Ring x) = CR (Multinomial x a)
  inl a = CR (MN (singleton empty a))
  inr (RingA (MN x)) = CR (MN (map initMN x))
  eva f g (CR c) = evalMN f (g · pvar) c
```

`eva f g (a + bx²y)` $\rightsquigarrow$ `f a` $\oplus$ `(f b` $\otimes$ `g x` $\otimes$ `g x` $\otimes$ `g y)`

Using the commutative ring free extension

inner product

```
[1; 0; 2] `dot` [⟦x⟧; ⟦y⟧; ⟦z⟧]
```

inner product, staged with ⟦ ⟧ and $:

```
⟦ (1 * x) + (0 * y) + (2 * z) ⟧
```

inner product, with partially-static data

```
⟦ x + (2 * z) ⟧
```

Lots more examples!
(see the paper)

Using *frex*

Using frex

1. write the instance

```
instance Monoid String where
  𝟙 = ""
  (⊛) = (++)
```

2. use frex's `Monoid (FreeExt`$_C$ `...)` instance:

```
(dyn x ⊛ sta "a") ⊛ (sta "b" ⊛ dyn x)
```

3. convert to code:

```
cd ((dyn x ⊛ sta "a") ⊛ (sta "b" ⊛ dyn x))
  ⇝   ⟦x ++ "ab" ++ x ⟧
```

Using frex with existing polymorphic code

```
dot :: Ring r ⇒ [r] → [r] → r
dot xs ys = sum (zipWith (×) xs ys)

mmmul :: Ring r ⇒ [[r]] → [[r]] → [[r]]
mmmul m n = [[dot a b | b ← transpose n] | a ← m]
```

Matrix multiplication unfolded

```
cdMtx $ staMtx (V (V 0 1) (V 1 2)) `mmmul` dynMtx ⟦m⟧
```

convert vectors to lists of partially-static values

```
cdMtx $ [[sta 1, sta 0],   `mmmul`  [[dyn ⟦m!0!0⟧, dyn ⟦m!0!1⟧],
         [sta 1, sta 2]]             [dyn ⟦m!1!0⟧, dyn ⟦m!1!1⟧]]
```

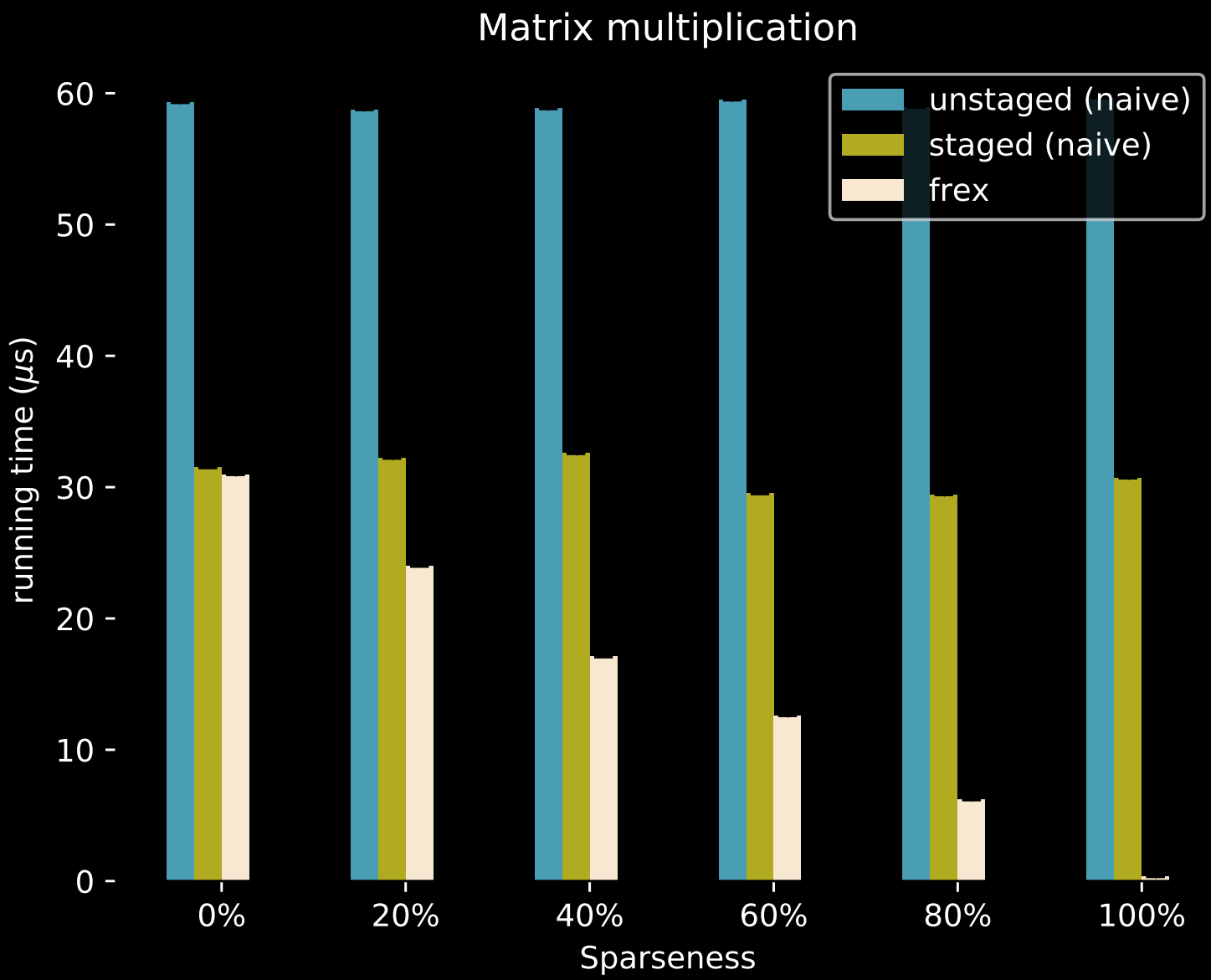
partially-static arithmetic with `mmmul`

```
cdMtx $ [[1×⟦m!0!0⟧ + 0×⟦m!1!0⟧, 1×⟦m!0!1⟧ + 0×⟦m!1!1⟧],
         [1×⟦m!0!0⟧ + 2×⟦m!1!0⟧, 1×⟦m!0!1⟧ + 2×⟦m!1!1⟧]]
```

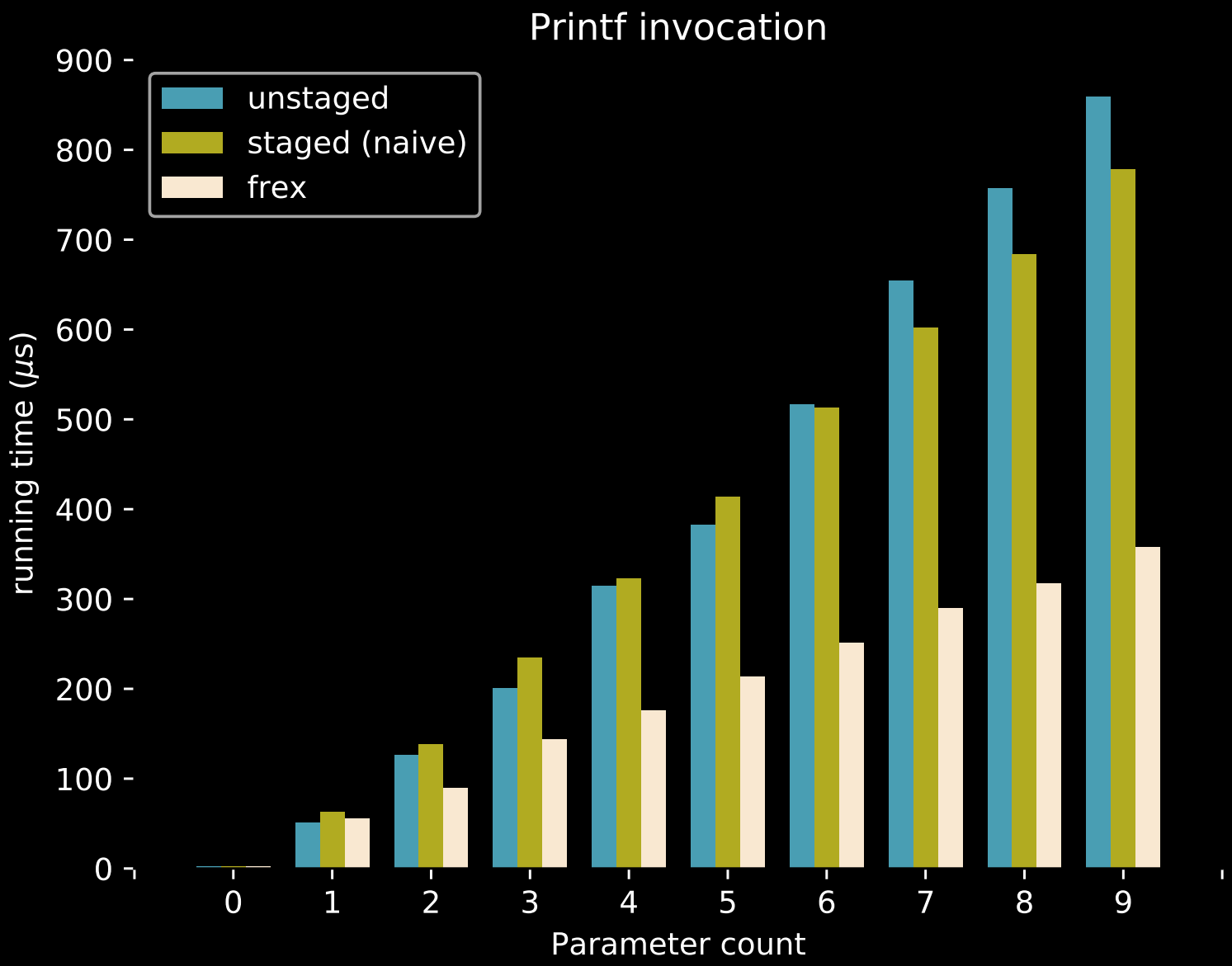
conversion to optimized code

```
⟦ V (V        m!0!0           m!0!1        )
    (V   m!0!0 + 2×m!1!0   m!0!1 + 2×m!1!1) ⟧
```

Performance improvements



Performance improvements



Summing up

1 a common pattern
(algebraic optimizations for staging)

2 a universal property
(partially-static data as free extensions)

3 a modular library
(

frex )