

# From binding signatures to monads in UniMath

**Anders Mörtberg** – Inria Sophia-Antipolis

Joint work with Benedikt Ahrens and Ralph Matthes

# Introduction

**Goal:** represent and reason about languages with binders using category theory in type theory

Start with a simple notion of signature representing a language with binders and from this construct a monad for this language

## Why monads?

-- A monad is a type family  $M$  with return and bind:

$\text{return} : a \rightarrow M a$

$(\gg=) : M a \rightarrow (a \rightarrow M b) \rightarrow M b$

-- We can define Kleisli composition for any monad:

$(\gg\Rightarrow) : (a \rightarrow M b) \rightarrow (b \rightarrow M c) \rightarrow (a \rightarrow M c)$

-- The monad laws can be written as:

$\text{return } a \gg= t = t a$

$t \gg= \text{return} = t$

$(t \gg= \sigma_1) \gg= \sigma_2 = t \gg= (\sigma_1 \gg\Rightarrow \sigma_2)$

## Why monads?

-- Substitution is a monad:

$\text{var} : a \rightarrow \text{Tm } a$

$\_[-] : \text{Tm } a \rightarrow (a \rightarrow \text{Tm } b) \rightarrow \text{Tm } b$

-- Kleisli composition is composition of substitutions:

$\_[-] ; \_[-] : (a \rightarrow \text{Tm } b) \rightarrow (b \rightarrow \text{Tm } c) \rightarrow (a \rightarrow \text{Tm } c)$

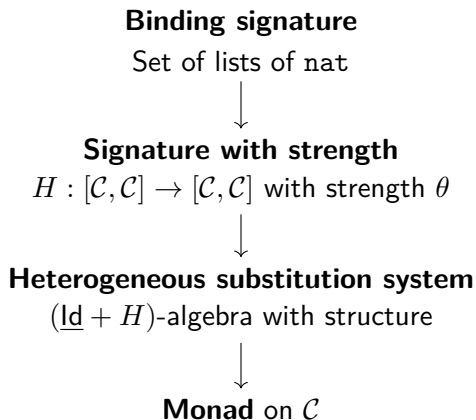
-- Monad laws are rules for substitution:

$(\text{var } a) [\sigma] = \sigma a$

$t [\lambda x \rightarrow \text{var } x] = t$

$(t [\sigma_1]) [\sigma_2] = t [\sigma_1 ; \sigma_2]$

# Overall structure



Formalized in UniMath: <https://github.com/UniMath/UniMath>

# UniMath: Univalent Mathematics

It is a core **language of dependent type theory**

- ▶ rich enough to formalize mathematics
- ▶ simple enough to allow for proof of consistency

## What UniMath has

Type former	Notation	(special case)
Inhabitant	$a : A$	
Dependent type	$x : A \vdash B(x)$	
Sigma type	$\sum_{(x:A)} B(x)$	$A \times B$
Product type	$\prod_{(x:A)} B(x)$	$A \rightarrow B$
Coproduct type	$A + B$	
Identity type	$\text{Id } A \ a \ b, \ a = b$	
Universe	$\mathbb{U}$	
<code>nat</code> , <code>bool</code> , <code>1</code> , <code>0</code>		

- ▶ Univalence axiom
- ▶ Consistent: simplicial and cubical set models

## Voevodsky's univalence axiom

**Univalence axiom:** equality of types is equivalent to equivalence of types

$$\text{univalence} : \text{Equiv } (A = B) (\text{Equiv } A B)$$

Univalence adds extensionality principles to intensional type theory:

- ▶ Function extensionality
- ▶ Propositional extensionality
- ▶ Set quotients
- ▶ Invariance under equivalence of types



# UniMath implementation

In practice, the UniMath language is a fragment of the Calculus of Inductive Constructions implemented in the Coq proof assistant with:

- ▶ Function extensionality and univalence added as axioms
- ▶ `Type` : `Type` (as a way to implement resizing)



# UniMath implementation

General purpose libraries:

- ▶ Foundations
- ▶ Number systems
- ▶ Algebra
- ▶ Category theory
- ▶ Homological algebra
- ▶ ...

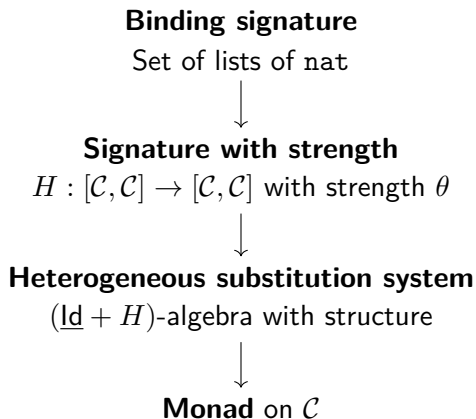
# What UniMath doesn't have

- ▶ General inductive types
- ▶ Record types
- ▶ Higher inductive types

In this talk I will describe a general framework for constructing various datatypes as initial algebras in UniMath

This means that inductive types do not have to be added to the core of UniMath, but can instead be justified in terms of the other notions

## Overall structure



# A simple notion of signature for variable binding

## Binding signature:

- ▶ A type  $I$  with decidable equality (“constructors”) and
- ▶ a function  $\text{arity} : I \rightarrow [\text{nat}]$

## Example: untyped lambda calculus

**Inductive** LC (X : Type) :=

| var : X -> LC X

| app : LC X \* LC X -> LC X

| abs : LC (option X) -> LC X

$I := \{\text{app}, \text{abs}\}$

$\text{arity}(\text{app}) = [0, 0]$

$\text{arity}(\text{abs}) = [1]$

# A categorical notion of signature for variable binding

## Signature with strength (Matthes & Uustalu)

- ▶ a functor  $H : [\mathcal{C}, \mathcal{C}] \rightarrow [\mathcal{C}, \mathcal{C}]$
- ▶ a natural transformation between bifunctors

$$\theta : (H-) \cdot U(\sim) \longrightarrow H(- \cdot U(\sim))$$

satisfying some axioms

Given  $(X, (Z, e))$  with  $X : [\mathcal{C}, \mathcal{C}]$  and  $(Z, e) : \text{Ptd}(\mathcal{C})$  we get:

$$\theta_{X, (Z, e)} : HX \cdot Z \rightarrow H(X \cdot Z)$$

# Untyped lambda calculus

The untyped lambda calculus as a binding signature:

$$I := \{\text{app}, \text{abs}\}$$
$$\text{arity}(\text{app}) = [0, 0]$$
$$\text{arity}(\text{abs}) = [1]$$

The untyped lambda calculus as a signature with strength:

- ▶  $H(F) := F \times F + F \cdot \text{option}$
- ▶  $\theta := \dots$

## From binding signatures to signatures with strength

Let  $(I, \text{arity})$  be a binding signature and  $i : I$ . To the list  $\text{arity}(i) = [n_1, \dots, n_k]$  we associate the functor:

$$[\mathcal{C}, \mathcal{C}] \rightarrow [\mathcal{C}, \mathcal{C}]$$
$$F \mapsto \prod_{1 \leq j \leq k} F \cdot \text{option}^{n_j}$$

The functor associated to the signature  $(I, \text{arity})$  is then obtained as the coproduct of the functors associated to each arity



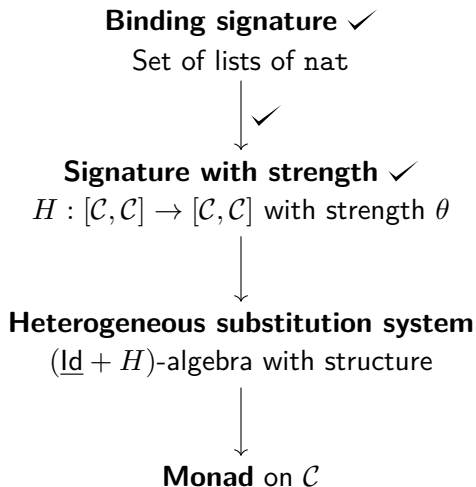
## From binding signatures to signatures with strength

$$H : [\mathcal{C}, \mathcal{C}] \rightarrow [\mathcal{C}, \mathcal{C}]$$
$$F \mapsto \prod_{i:I} \prod_{1 \leq j \leq \text{length}(\text{arity}(i))} F \cdot \text{option}^{\text{arity}(i)_j}$$

For details on how to construct  $\theta$  see the paper

We want to instantiate this with  $\mathcal{C} = \text{Set}$ , for this  $I$  has to be a set

## Recall: overall structure



# Heterogeneous substitution system

## Definition (Matthes & Uustalu)

Let  $(H, \theta)$  be a signature with strength. A **heterogeneous substitution system** (hss) is a  $(\underline{\text{Id}} + H)$ -algebra  $(T, \alpha)$  with some extra structure

This means

$$\alpha : (\underline{\text{Id}} + H)T \rightarrow T$$

which gives two natural transformations  $\eta : \text{Id} \rightarrow T$  and  $\tau : HT \rightarrow T$

$\eta_C : C \rightarrow TC$  is the injection of variables and  $\tau$  represents all the other constructors of the language

# Heterogeneous substitution systems and monads

Theorem (Matthes & Uustalu, formalized by Ahrens & Matthes)

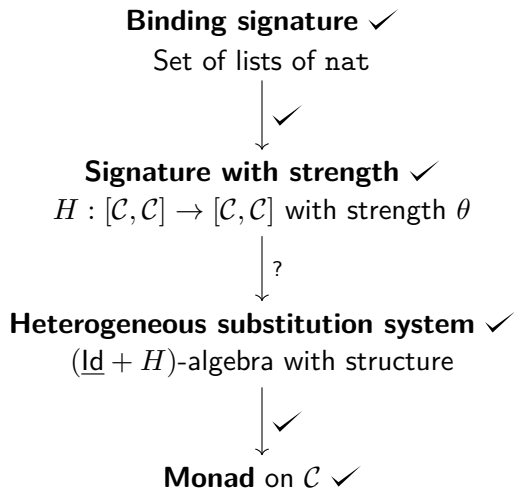
*If  $(T, \alpha)$  is a hss for  $(H, \theta)$  then  $T$  is a monad with  $\eta$  as unit and join defined using the extra structure of  $(T, \alpha)$*

Theorem (Ahrens, Matthes & M.)

*If  $H$  is  $\omega$ -cocontinuous then we can construct a hss  $(T, \alpha)$  as the initial algebra of  $(\text{Id} + H)$*

This is a variation of a previous result of Matthes & Uustalu which required the existence of a particular right adjoint which made it not applicable to Set

## Recall: overall structure



## Datatypes as initial algebras

Need to construct initial algebra for  $(\underline{\text{Id}} + H) : [\mathcal{C}, \mathcal{C}] \rightarrow [\mathcal{C}, \mathcal{C}]$

We do this for general endofunctors  $F : \mathcal{D} \rightarrow \mathcal{D}$  satisfying some conditions

From this we can construct many **inductive types** in UniMath, not only those representing language with binders (e.g. lists, trees...)

# Formal definition of inductive types

## What are inductive types?

Two characterizations:

**external** via inference rules

**internal** via universal property – as initial algebra

## Our goal

We are interested in internally characterized inductive types, and their construction in the UniMath language

## Why the need for a systematic construction?

**Some** inductive types are easily constructed, e.g., lists over a given base type:

- ▶  $\text{Vect}(A, n) := A^n$
- ▶  $[A] := \sum_{(n:\text{nat})} \text{Vect}(A, n)$

But it is not always that easy

### Exercise

Define an equivalent type to LC from above using just the UniMath language



## Datatypes categorically: lists of sets

In order to construct lists of over a set  $A$  we start with the list functor:

$$L_A(X) = 1 + A \times X$$

Assuming that we can construct initial algebras we get

$$\mu L_A : \text{Set} \qquad \alpha : 1 + A \times \mu L_A \rightarrow \mu L_A$$

From  $\alpha$  we get the constructors:

$$\text{nil} : \mu L_A \qquad \text{cons} : A \rightarrow \mu L_A \rightarrow \mu L_A$$

## Datatypes categorically: lists of sets

As  $(\mu L_A, \alpha)$  is initial we get for any set  $X$ , element  $x : X$  and  $f : A \times X \rightarrow X$  a unique function  $\text{foldr} : \mu L_A \rightarrow X$  satisfying:

$$\begin{array}{ccc} 1 + A \times \mu L_A & \xrightarrow{\alpha} & \mu L_A \\ L_A(\text{foldr}) \downarrow & & \downarrow \text{foldr} \\ 1 + A \times X & \xrightarrow{[\lambda_.x, f]} & X \end{array}$$

That is,

$$\begin{aligned} \text{foldr nil} &= x \\ \text{foldr (cons } y \text{ } ys) &= f (y, \text{foldr } ys) \end{aligned}$$

# Construction of initial algebras in UniMath

Initial algebra of  $F : \mathcal{C} \rightarrow \mathcal{C}$  (Adámek)

If  $F$  is  $\omega$ -cocontinuous, then the colimit of

$$0 \rightarrow F0 \rightarrow F^2 0 \rightarrow \dots$$

is an initial  $F$ -algebra

We hence need:

- ▶ Initial object,  $0 : \mathcal{C}$
- ▶ Colimits of chains in  $\mathcal{C}$
- ▶ Proof that  $F$  is  $\omega$ -cocontinuous

# Construction of initial algebras in UniMath

If  $\mathcal{C} = \text{Set}$  we can easily prove that the empty set is initial, but what about colimits?

Colimits can be constructed from coproducts and coequalizers:

- ▶ in plain type theory we have coproducts
- ▶ in univalent type theory, additionally have set quotients  
a.k.a. coequalizers in  $\text{Set}$

## Restriction

This approach only allows construction of inductive **sets**

# Set quotients in UniMath

Voevodsky has defined set quotients  $X/R$  for an equivalence relation  $R : X \rightarrow X \rightarrow \mathbf{hProp}$

This construction uses function extensionality and univalence for propositions

It also uses an impredicative encoding of propositional truncation:

$$\|A\| := \Pi_{(P:\mathbf{hProp})}(A \rightarrow P) \rightarrow P$$

which requires propositional resizing

## $\omega$ -cocontinuous functors

For the example of lists we need to prove that  $L_A(X) = 1 + A \times X$  is  $\omega$ -cocontinuous

We can write this “point-free” as:  $L_A = \underline{1} + A \times \_$

So we need to prove that the following functors are  $\omega$ -cocontinuous:

- ▶ Constant functor
- ▶ Sum of functors ( $F + G : \text{Set} \rightarrow \text{Set}$ )
- ▶ Product with a fixed element ( $A \times \_ : \text{Set} \rightarrow \text{Set}$ )

All of these are straightforward (using that left adjoints preserve colimits)

## $\omega$ -cocontinuous functors

Recall that we want to construct initial algebras for the functor:

$$H : [\mathcal{C}, \mathcal{C}] \rightarrow [\mathcal{C}, \mathcal{C}]$$
$$F \mapsto \prod_{i:I} \prod_{1 \leq j \leq \text{length}(\text{arity}(i))} F \cdot \text{option}^{\text{arity}(i)_j}$$

For this we also need that the following functors are  $\omega$ -cocontinuous:

- ▶ Coproducts of a family of functors
- ▶ Product of functors
- ▶ Precomposition with option

These are **a lot** more difficult!

## $\omega$ -cocontinuous functors: product of functors

**Key lemma:** The functor  $\times : \mathcal{C}^2 \rightarrow \mathcal{C}$  is  $\omega$ -cocontinuous for  $\mathcal{C}$  cartesian closed

Proof idea: Given a diagram

$$(A_0, B_0) \xrightarrow{(f_0, g_0)} (A_1, B_1) \xrightarrow{(f_1, g_1)} (A_2, B_2) \xrightarrow{(f_2, g_2)} \dots$$

with colimit  $(L, R)$ , we need to show that  $L \times R$  is the colimit of

$$A_0 \times B_0 \xrightarrow{f_0 \times g_0} A_1 \times B_1 \xrightarrow{f_1 \times g_1} A_2 \times B_2 \xrightarrow{f_2 \times g_2} \dots$$



## $\omega$ -cocontinuous functors: product of functors

To this end, we consider the grid

$$\begin{array}{ccccccc} (A_0, B_0) & \xrightarrow{(f_{0,1})} & (A_1, B_0) & \xrightarrow{(f_{1,1})} & (A_2, B_0) & \xrightarrow{(f_{2,1})} & \dots \\ (1, g_0) \downarrow & & (1, g_0) \downarrow & & \downarrow & & \\ (A_0, B_1) & \xrightarrow{(f_{0,1})} & (A_1, B_1) & \xrightarrow{(f_{1,1})} & (A_2, B_1) & \xrightarrow{(f_{2,1})} & \dots \\ (1, g_1) \downarrow & & \downarrow & & \downarrow & & \\ \vdots & & \vdots & & \vdots & & \vdots \end{array}$$

## $\omega$ -cocontinuous functors: product of functors

Proof idea is simple, but formalization hard because the type of the arrows involves a lot of index manipulations:

**Definition** fun\_lt (cAB : chain (C \* C)) :

$$\prod i j, i < j \rightarrow C[\text{ob1 (dob cAB i)} \times \text{ob2 (dob cAB j)}, \\ \text{ob1 (dob cAB j)} \times \text{ob2 (dob cAB j)}].$$

**Proof.**

intros i j hij.

apply (BinProductOfArrows (chain\_mor cAB hij) (identity \_)).

Defined.

**Definition** map\_to\_K (cAB : chain (C \* C)) (K : C)

$$(\text{cck} : \text{cocone (mapchain (\times) cAB) K}) i j : \\ C[\text{ob1 (dob cAB i)} \times \text{ob2 (dob cAB j)}, K].$$

**Proof.**

destruct (natlthorgeh i j) as [Hlt|Hge].

- apply (fun\_lt cAB \_ \_ Hlt ;; coconeIn cck j).

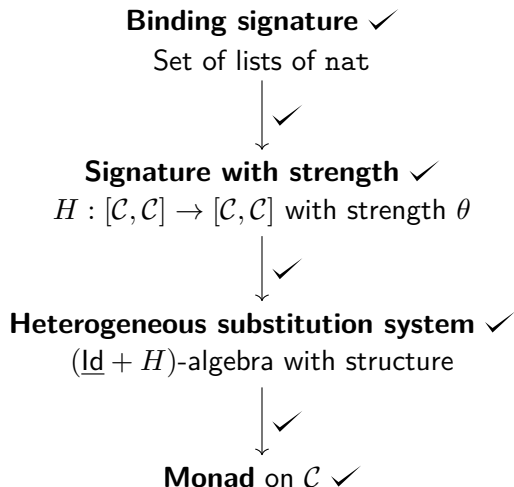
- destruct (natgehchoice \_ \_ Hge) as [Hlt|Heq].

+ apply (fun\_gt cAB \_ \_ Hlt ;; coconeIn cck i).

+ destruct Heq; apply (coconeIn cck i).

Defined.

## Recall: overall structure



# From signatures to monads in UniMath

We have defined the following function in UniMath:

**Definition** `BindingSigToMonad` :

```
II (C : Precategory) (BPC : BinProducts C),
  BinCoproducts C → Terminal C → Initial C
→ Colims_of_shape nat_graph C
→ (II F, is_omega_cocont (constprod_functor1 F))
→ II sig : BindingSig, Products (BindingSigIndex sig) C
→ Coproducts (BindingSigIndex sig) C
→ Monad C.
```

All of the hypotheses are fulfilled by `Set`:

**Definition** `BindingSigToMonadHSET` : `BindingSig` → `Monad HSET`.

## Example: untyped lambda calculus

The untyped lambda calculus is represented by the binding signature:

$$\begin{aligned} I &:= \{\text{app}, \text{abs}\} \\ \text{arity}(\text{app}) &= [0, 0] \\ \text{arity}(\text{abs}) &= [1] \end{aligned}$$

This is easily implemented in UniMath:

**Definition** LamSig : BindingSig :=  
mkBindingSig isdeceqbool ( $\lambda$  b, if b then [0,0] else [1]).

**Definition** LamMonad : Monad HSET := BindingSigToMonadHSET LamSig.

## Constructive mathematics and computer programming†

BY P. MARTIN-LÖF

*Department of Mathematics, University of Stockholm, Box 6701, S-113 85 Stockholm, Sweden*

If programming is understood not as the writing of instructions for this or that computing machine but as the design of methods of computation that it is the computer's duty to execute (a difference that Dijkstra has referred to as the difference between computer science and computing science), then it no longer seems possible to distinguish the discipline of programming from constructive mathematics. This explains why the intuitionistic theory of types (Martin-Löf 1975 In *Logic Colloquium 1973* (ed. H. E. Rose & J. C. Shepherdson), pp. 73–118. Amsterdam: North-Holland), which was originally developed as a symbolism for the precise codification of constructive mathematics, may equally well be viewed as a programming language. As such it provides a precise notation not only, like other programming languages, for the programs themselves but also for the tasks that the programs are supposed to perform. Moreover, the inference rules of the theory of types, which are again completely formal, appear as rules of correct program synthesis. Thus the correctness of a program written in the theory of types is proved formally at the same time as it is being synthesized.

During the period of just over thirty years that has elapsed since the first electronic computers were built, programming languages have developed from various machine codes and assembly languages, now referred to as low level languages, to high level languages, like FORTRAN, ALGOL 60 and 68, LISP and PASCAL. The virtue of a machine code is that a program written in it can be directly read and executed by the computer.

Types	Concrete syntax	Binding arities
Pi types	$(\Pi x:A)B, (\lambda x)b, (c)a$	$[0,1], [1], [0,0]$
Sigma types	$(\Sigma x:A)B, (a,b), (E x,y)(c,d)$	$[0,1], [0,0], [0,2]$
Sum types	$A + B, i(a), j(b), (D x,y)(c,d,e)$	$[0,0], [0], [0], [0,1,1]$
Id types	$I(A,a,b), r, J(c,d)$	$[0,0,0], [], [0,0]$
Fin types	$N_i, 0_i \cdots (i-1)_i, R_i(c, c_0, \dots, c_{i-1})$	$[], [] \cdots [], [0,0, \dots, 0]$
Natural numbers	$\mathbb{N}, 0, a', (R x,y)(c,d,e)$	$[], [], [0], [0,0,2]$
W-types	$(W x \in A)B, \text{sup}(a,b), (T x,y,z)(c,d)$	$[0,1], [0,0], [0,3]$
Universes	$U_0, U_1, \dots$	$[], [], \dots$

This is an example of a language with infinitely many constructors

## MLTT79 in UniMath

**Definition** PiSig : BindingSig :=  
mkBindingSig (isdeceqstn 3) (three\_rec [0,1] [1] [0,0]).

**Definition** SigmaSig : BindingSig :=  
mkBindingSig (isdeceqstn 3) (three\_rec [0,1] [0,0] [0,2]).

...

**Definition** USig : BindingSig := mkBindingSig isdeceqnat ( $\lambda \_ , []$ ).

**Definition** MLTT79Sig := PiSig ++ SigmaSig ++ SumSig ++ IdSig ++  
FinSig ++ NatSig ++ WSig ++ USig.

**Definition** MLTT79Monad : Monad HSET :=  
BindingSigToMonadHSET MLTT79Sig.



# Summary

We have formalized:

- ▶ Translation from binding signatures to monads
- ▶ Examples: untyped lambda calculus and MLTT79
- ▶ General framework for constructing datatypes as initial algebras in UniMath

We have used function extensionality and univalence for propositions which both had to be added as axioms to Coq... Computation?

## Example: lists in UniMath

**Definition** `length` : `List A`  $\rightarrow$  `nat` :=  
 `foldr natHSET 0 ( $\lambda$  _ (n : nat), 1 + n)`.

**Eval lazy in** `length (5 :: 2 :: [])`.  
> `2` : `nat`

**Eval compute in** `length (5 :: 2 :: [])`.  
> ...

**Eval lazy in** `[]`.  
> ...

## Example: lists in UniMath

**Lemma** `foldr_nil` ( $X : \text{hSet}$ ) ( $x : X$ ) ( $f : A \rightarrow X \rightarrow X$ ) :  
 `foldr X x f nil = x`.

**Lemma** `foldr_cons` ( $X : \text{hSet}$ ) ( $x : X$ ) ( $f : A \rightarrow X \rightarrow X$ ) ( $a : A$ ) ( $l : \text{List } A$ ) :  
 `foldr X x f (cons a l) = f a (foldr X x f l)`.

**Lemma** `listIndhProp` ( $P : \text{List } A \rightarrow \text{hProp}$ ) :  
 `P nil  $\rightarrow$  ( $\prod a l, P l \rightarrow P (\text{cons } a l)$ )  $\rightarrow \prod l, P l$` .

**Lemma** `length_map` ( $f : A \rightarrow A$ ) :  $\prod xs, \text{length } (\text{map } f xs) = \text{length } xs$ .

**Proof.**

`apply listIndProp; simpl.`

- `apply idpath.`

- `unfold length, map; intros a l IH.`

`now rewrite !foldr_cons, <- IH.`

**Qed.**

## Future goals

- ▶ Multisorted signatures (STLC, System F...)
- ▶ Show that the datatype together with the constructed substitution operation is initial in a category of “algebras with substitution”
- ▶ Connect to Voevodsky’s work on C-systems and models of type theory

# Thank you for your attention!

<https://arxiv.org/abs/1612.00693>