Recursion Schemes in Coq

APLAS '19@Bali

2019/12/02

Kosuke MURATA Kento Emoto

Kyushu Institute of Technology, Japan

Our Contribution

- A Coq library for **datatype-generic** program calculation
  - based on [Tesson+, 2011]
  - This library enable users to write the datatype-generic proof in the **equational style**
- We give Coq proofs of all theorems in [Uustalu+, 1999]
  - Their work describes formal definitions and calculation rules for various kinds of recursion schemes in category **Set**
  - They propose histomorphisms and futumorphisms

Outline

1. Introduction
2. Recursion Schemes
3. Coq library
4. Conclusion

Program Calculation

... is a programing technique for deriving fast and highly-technical programs from naive ones

Naive program (algorithm)

equational reasoning
(applying **calculation rules**)

Faster program (alogorithm)

A Core Technique of Program Calculation:

Equational Reasoning for Programs

$$
\begin{aligned}
& \underline{mss} \\
= & \ \{ \text{ the def. of } mss \ \} \\
& max \circ \underline{map\ sum \circ concat} \circ map\ tails \circ inits \qquad \#\ O(n^3) \\
= & \ \{ \ concat\text{-}map\text{-fusion} \ \} \\
& max \circ \underline{concat \circ map\ (map\ sum)} \circ map\ tails \circ inits \\
= & \ \{ \ concat\text{-}foldr\text{-fusion} \ \} \\
& max \circ \underline{map\ (map\ max) \circ map\ (map\ sum) \circ map\ tails} \circ inits \\
= & \ \{ \ map\text{-}map\text{-sufion (twice)} \ \} \\
& max \circ map\ (map\ max \circ map\ sum \circ tails) \circ inits \\
& \vdots \\
= & \ \ max \circ map\ (fst \circ foldr\ (\odot)\ e' \circ map\ f) \circ inits \qquad \#\ O(n) \\
& \ \ \textbf{where} \quad f\ a = (a, a),\ e' = (-\infty, 0), \\
& \qquad\qquad (m_1, s_1) \odot (m_2, s_2) = ((m_1 + s_2) \uparrow m_2,\ s_1 + s_2)
\end{aligned}
$$

A Core Technique of Program Calculation: Equational Reasoning for Programs

naive program

$$
\begin{aligned}
& \underline{mss} \\
= & \{ \text{ the def. of } mss \text{ } \} \\
& max \circ \underline{map\ sum \circ concat} \circ map\ tails \circ inits \qquad \#\ O(n^3) \\
= & \{ \ concat\text{-}map\text{-fusion } \} \\
& max \circ \underline{concat \circ map\ (map\ sum)} \circ map\ tails \circ inits \\
= & \{ \ concat\text{-}foldr\text{-fusion } \} \\
& max \circ \underline{map\ (map\ max) \circ map\ (map\ sum) \circ map\ tails} \circ inits \\
= & \{ \ map\text{-}map\text{-sufion (twice) } \} \\
& max \circ map\ (map\ max \circ map\ sum \circ tails) \circ inits \\
& \vdots \\
= & \ max \circ map\ (fst \circ foldr\ (\odot)\ e' \circ map\ f) \circ inits \qquad \#\ O(n) \\
& \ \textbf{where} \quad f\ a = (a, a),\ e' = (-\infty, 0), \\
& \qquad\qquad (m_1, s_1) \odot (m_2, s_2) = ((m_1 + s_2) \uparrow m_2,\ s_1 + s_2)
\end{aligned}
$$

A Core Technique of Program Calculation: Equational Reasoning for Programs

$$
\begin{aligned}
& \underline{mss} \\
= & \ \{ \text{the def. of } mss \} \\
& max \circ \underline{map\ sum \circ concat} \circ map\ tails \circ inits \qquad \#\ O(n^3) \\
= & \ \{ concat\text{-}ma\ldots \} \\
& max \circ co\ldots\ map\ tails \circ inits \\
= & \ \{ concat\text{-}foldr\text{-fusion} \} \\
& max \circ \underline{map\ (map\ max) \circ map\ (map\ sum) \circ map\ tails} \circ inits \\
= & \ \{ map\text{-}map\text{-sufion (twice)} \} \\
& max \circ map\ (map\ max \circ map\ sum \circ tails) \circ inits \\
& \vdots \\
= & \ max \circ map\ (fst \circ foldr\ (\odot)\ e' \circ map\ f) \circ inits \qquad \#\ O(n) \\
& \textbf{where} \quad f\ a = (a, a),\ e' = (-\infty, 0), \\
& \qquad (m_1, s_1) \odot (m_2, s_2) = ((m_1 + s_2) \uparrow m_2,\ s_1 + s_2)
\end{aligned}
$$

naive program

applying caluclation laws

A Core Technique of Program Calculation:
Equational Reasoning for Programs

naive program

$$
\begin{aligned}
& \underline{mss} \\
= & \ \{ \text{ the def. of } mss \ \} \\
& max \circ \underline{map\ sum \circ concat} \circ map\ tails \circ inits \qquad \#\ O(n^3) \\
= & \ \{ \ concat\text{-}ma \ldots \\
& max \circ co \ldots \ map\ tails \circ inits \\
= & \ \{ \ concat\text{-}foldr\text{-fusion} \ \} \\
& max \circ \underline{map\ (map\ max) \circ map\ (map\ sum) \circ map\ tails} \circ inits \\
= & \ \{ \ map\text{-}map\text{-sufion (twice)} \ \} \\
& max \circ map\ (map\ max \circ map\ sum \circ tails) \circ inits \\
& \vdots \\
= & \ max \circ map\ (fst \circ foldr\ (\odot)\ e' \circ map\ f) \circ inits \qquad \#\ O(n) \\
& \ \textbf{where} \quad f\ a = (a, a),\ e' = (-\infty, 0), \\
& \qquad \ldots ((m_1 + s_2) \uparrow m_2,\ s_1 + s_2)
\end{aligned}
$$

applying caluclation laws

deriving faster program!

An example of calculation rules

Map fusion law

- famous calculation rule for list functions:

$$(\mathbf{map}\ g) \circ (\mathbf{map}\ f) = \mathbf{map}\ (g \circ f)$$

An example of calculation rules
Map fusion law

- famous calculation rule for list functions:

$$(\mathbf{map}\ g) \circ (\mathbf{map}\ f) = \mathbf{map}\ (g \circ f)$$

$[a_1, \ldots, a_n]$

An example of calculation rules

Map fusion law

- famous calculation rule for list functions:

$$(\mathbf{map}\ g) \circ (\mathbf{map}\ f) = \mathbf{map}\ (g \circ f)$$

$$[a_1, \ldots, a_n] \xmapsto{\mathbf{map}\ f} [f\ a_1, \ldots, f\ a_n]$$

An example of calculation rules

Map fusion law

- famous calculation rule for list functions:

$$(\textbf{map}\ g) \circ (\textbf{map}\ f) = \textbf{map}\ (g \circ f)$$

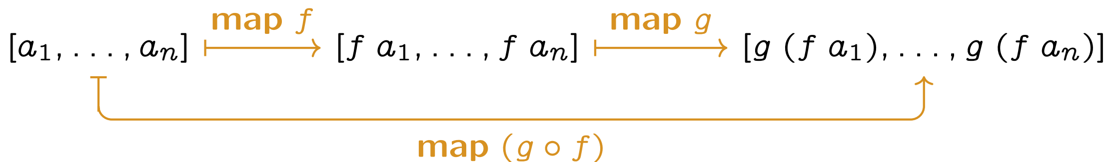
$$[a_1, \ldots, a_n] \xmapsto{\textbf{map}\ f} [f\ a_1, \ldots, f\ a_n] \xmapsto{\textbf{map}\ g} [g\ (f\ a_1), \ldots, g\ (f\ a_n)]$$

An example of calculation rules

Map fusion law

- famous calculation rule for list functions:

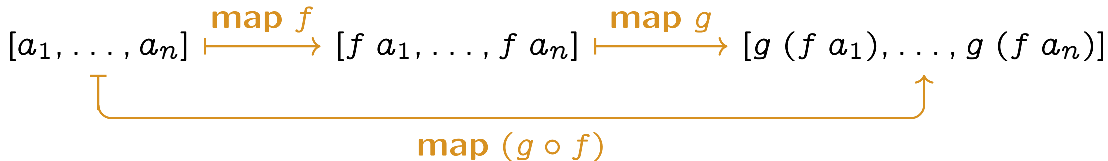
$$(\textbf{map}\ g) \circ (\textbf{map}\ f) = \textbf{map}\ (g \circ f)$$



An example of calculation rules
Map fusion law

- famous calculation rule for list functions:

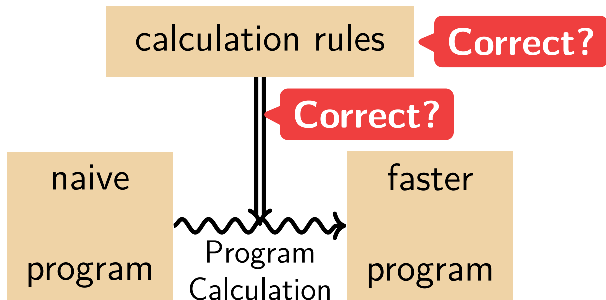
$$(\mathbf{map}\ g) \circ (\mathbf{map}\ f) = \mathbf{map}\ (g \circ f)$$



- usually RHS is faster than LHS
  - because RHS makes no intermediate data structures

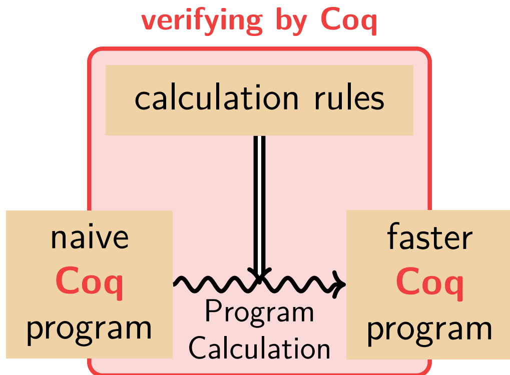
Certified Program Calculation

- Is the calculation really "correct"?



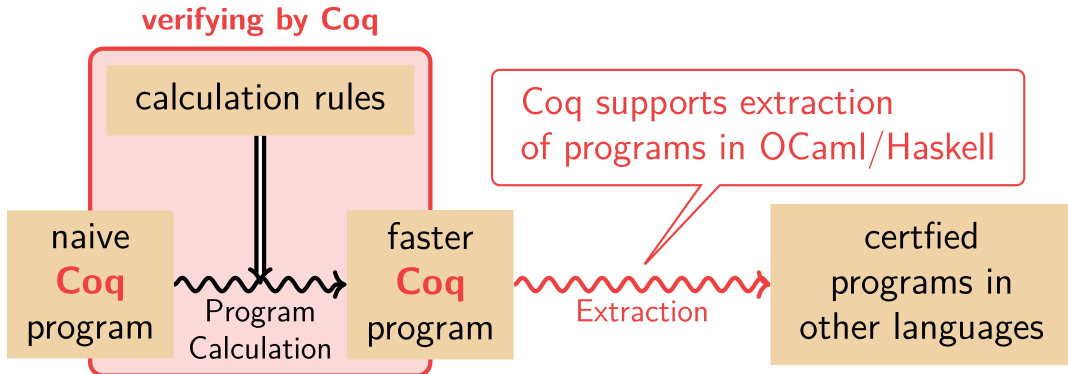
Certified Program Calculation

- Is the calculation really "correct"? $\longrightarrow$ **Let's verify by Coq!**



Certified Program Calculation

- Is the calculation really "correct"? → **Let's verify by Coq!**



- Many papers of program calculation adopts equational-style proof to prove calculation rules
  - Translating into Coq script from equational-style proof is not easy

an equational-style proof

$$
\begin{aligned}
& \mathbf{map}\,(g \circ f)\,[\,] \\
= & \ \{ \text{ the def. of } \mathbf{map} \ \} \\
& (\mathbf{map}\ g \circ \mathbf{map}\ f)\,[\,]
\end{aligned}
$$

$$
\begin{aligned}
& \mathbf{map}\,(g \circ f)\,(x :: xs) \\
= & \ \{ \text{ the def. of } \mathbf{map} \ \} \\
& (g\,(f\,x)) :: (\mathbf{map}\,(g \circ f)\,xs) \\
= & \ \{ \text{ induction hypo. } \} \\
& (g\,(f\,x)) :: ((\mathbf{map}\ g \circ \mathbf{map}\ f)\,xs) \\
= & \ \{ \text{ the def. of } \mathbf{map} \ \} \\
& (\mathbf{map}\ g \circ \mathbf{map}\ f)\,(x :: xs)
\end{aligned}
$$

a Coq scripts

```
intros ? ? ? ? ? xs.
induction xs as [|? ? IHx].
- reflexivity.
- cbn.
  rewrite IHx.
  reflexivity.
```

(both are proofs of the map-fusion law)

- If there is a library to support equational-style notation in Coq,
  - what you need to prove calculation rules:
    simply **"copy" the proof in the papers**!

an equational-style proof

$$
\begin{aligned}
& \mathbf{map}\,(g \circ f)\,[\,] \\
= & \ \{ \text{ the def. of } \mathbf{map} \ \} \\
& (\mathbf{map}\,g \circ \mathbf{map}\,f)\,[\,]
\end{aligned}
$$

$$
\begin{aligned}
& \mathbf{map}\,(g \circ f)\,(x :: xs) \\
= & \ \{ \text{ the def. of } \mathbf{map} \ \} \\
& (g\,(f\,x)) :: (\mathbf{map}\,(g \circ f)\,xs) \\
= & \ \{ \text{ induction hypo. } \} \\
& (g\,(f\,x)) :: ((\mathbf{map}\,g \circ \mathbf{map}\,f)\,xs) \\
= & \ \{ \text{ the def. of } \mathbf{map} \ \} \\
& (\mathbf{map}\,g \circ \mathbf{map}\,f)\,(x :: xs)
\end{aligned}
$$

a Coq scripts

```
intros ? ? ? ? ? xs.
induction xs as [|? ? IHx].
Left
= Right.
Left
= ((g (f x))::(map (g ∘ f)) xs).
= ((g (f x))::(map g ∘ map f) xs)
  { by IHx }.
= Right.
```

Previous Work[Tesson+, ‘11]

- a Coq tactic library for program calculation
  - enable users to write equational-style proof
- They formalize “Theory of Lists” using their library
  - a lecture note by Bird.
  - providing a set of calculation rules for list functions
- adopts **shallow embedding**
  - users can write proofs about runnable Coq programs directly

```
Theorem filter_promotion : p <| :o: @concat A = @concat A :o: p <| *.
Proof.
    LHS
= { rewrite (filter mapreduce) }
    (++ / :o: f * :o: @concat A).
= { rewrite map promotion }
    (++ /:o: @concat (list A) :o: f* *).
= { rewrite comp assoc }
    ((++ /:o: @concat (list A)) :o: f* *).
= { rewrite reduce promotion }
    (++ / :o: (++ /)* :o: f* *).
= { rewrite concat reduce }
    (@concat A :o: (++ /)* :o: f* *).
= { rewrite map distr comp }
    (@concat A :o: (++ / :o: f *) *).
= { rewrite filter mapreduce }
    (@concat A :o: (p <|) *).
[]. Qed.
```

This study

Generalizing to any datatypes!

- Applications of their library are limited to list functions
- but functional programers use various datatypes
  - not only the lists, but also the natural numbers, binary trees, or other user-defiend datatypes (ADT; Algebraic Data Types)
- This study: **we generalize the library to datatype-generic calculation**!

This study

Generalizing to any datatypes!

- Applications of their library are limited to list functions
- but functional programers use various datatypes
  - not only the lists, but also the natural numbers, binary trees, or other user-defiend datatypes (ADT; Algebraic Data Types)
- This study: **we generalize the library to datatype-generic calculation**!

Our library based on **Recursion Schemes!**

Outline

1. Introduction
2. Recursion Schemes
3. Coq library
4. Conclusion

Recursion Schemes

- are generalized **fold** and **unfold** operators
  - **fold** tears down an input data-structure
  - **unfold** builds up output data-structure
  - They are parametrized by datatypes!
- well-studied in the context of datatype-generic programing

Recursion Schemes

- are generalized **fold** and **unfold** operators
  - **fold** tears down an input data-structure
  - **unfold** builds up output data-structure
  - They are parametrized by datatypes!
- well-studied in the context of datatype-generic programing
- using simple categorical notions
  - in this study, we work on **Set**
  - because Coq functions are total

Datatypes by Categorical Notions

- Inductive/coinductive datatypes can be modeled by initial/terminal (co)algebras of a endofunctor
  - $N(X) = \mathbf{1} + X$
    - μN: natural numbers

Datatypes by Categorical Notions

- Inductive/coinductive datatypes can be modeled by initial/terminal (co)algebras of a endofunctor
  - $N(X) = \mathbf{1} + X$
    - μN: natural numbers
  - $L_A(X) = \mathbf{1} + A \times X$
    - μL_A: finite lists of A
    - νL_A: finite/infinite lists of A

Datatypes by Categorical Notions

- Inductive/coinductive datatypes can be modeled by initial/terminal (co)algebras of a endofunctor
  - $N(X) = \mathbf{1} + X$
    - μN: natural numbers
  - $L_A(X) = \mathbf{1} + A \times X$
    - μL_A: finite lists of A
    - νL_A: finite/infinite lists of A
  - $T_A(X) = \mathbf{1} + A \times X \times A$
    - μT_A: finite trees of A
    - νT_A: finite/infinite trees of A

Folding

(tearing down the inductive datatypes)

Catamorphism[Malcom, 1990]

$$f = \varphi \circ F(f) \circ \mathbf{in}_F^{-1}$$

Paramorphism[Meertens, 1992]

$$f = \varphi \circ F(\langle f, \mathrm{id}_{\mu F} \rangle) \circ \mathbf{in}_F^{-1}$$

Histomorphism[Uustalu+, 1999]

$$f = \varphi \circ F([\![\langle f, \mathbf{in}_F^{-1} \rangle]\!]) \circ \mathbf{in}_F^{-1}$$

Unfolding

(building up the coinductive datatypes)

Anamorphism[Fokkinga+, 1991]

$$f = \mathbf{out}_F^{-1} \circ F(f) \circ \psi$$

Apomorphism[Vos, 1995]

$$f = \mathbf{out}_F^{-1} \circ F([f, \mathrm{id}_{\nu F}]) \circ \psi$$

Futumorphism[Uustalu+, 1999]

$$f = \mathbf{out}_F^{-1} \circ F((\!| [f, \mathbf{out}_F^{-1}] |\!)) \circ \psi$$

Refolding

(Building up an intermediate data structure then tearing down it)

Hylomorphism[Meijer+, 1991]

$$cata \circ ana$$

Dynamorphism[Kabanov+, 2006]

$$histo \circ ana$$

Folding

(tearing down the inductive datatypes)

Catamorphism[Malcom, 1990]

$$f = \varphi \circ F(f) \circ \mathbf{in}_F^{-1}$$

Paramorphism[Meertens, 1992]

$$f = \varphi \circ F(\langle f, \mathrm{id}_{\mu F} \rangle) \circ \mathbf{in}_F^{-1}$$

Histomorphism[Uustalu+, 1999]

$$f = \varphi \circ F(\llbracket \langle f, \mathbf{in}_F^{-1} \rangle \rrbracket) \circ \mathbf{in}_F^{-1}$$

Unfolding

(building up the coinductive datatypes)

Anamorphism[Fokkinga+, 1991]

$$f = \mathbf{out}_F^{-1} \circ F(f) \circ \psi$$

Apomorphism[Vos, 1995]

$$f = \mathbf{out}_F^{-1} \circ F([f, \mathrm{id}_{\nu F}]) \circ \psi$$

Futumorphism[Uustalu+, 1999]

$$f = \mathbf{out}_F^{-1} \circ F(\llparenthesis [f, \mathbf{out}_F^{-1}] \rrparenthesis) \circ \psi$$

Refolding

(Building up an intermediate data structure then tearing down it)

Hylomorphism[Meijer+, 1991]

$$cata \circ ana$$

Dynamorphism[Kabanov+, 2006]

$$histo \circ ana$$

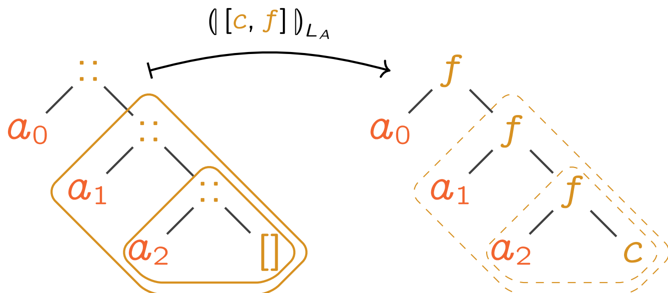
Catamorphisms

- simplest generalized-**fold** operator
  - tears down an input value by replacing constructors with (same-type) given functions
- denoted like $(\!|\, \varphi \,|\!)_F$

An example for lists:

$$L_A(X) = \mathbf{1} + A \times X$$

$$\begin{cases} c & : & \mathbf{1} \to X \\ f & : & A \times X \to X \end{cases}$$

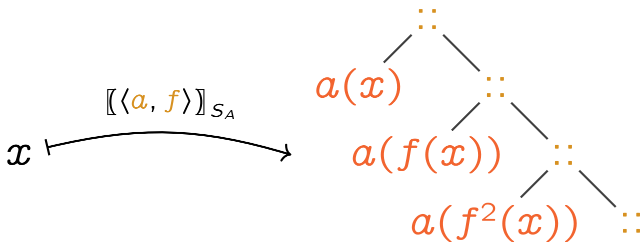


Anamorphisms

- simplest generalized-**unfold** operator
  - builds up a output value using given functions
- denoted like $[\!(\varphi)\!]_F$

An example for streams:

$$S_A(X) = A \times X$$
$$a : X \to A$$
$$f : X \to X$$



Datatype-generic map

- A **generalized map** can be defined by using catamorphism:

$$\mathbf{map}_F\ f = (\!|\, \mathbf{in}_F \circ F(f,\ \mathrm{id})\, |\!)$$

  - A bifunctor F decides the shape of trees

Datatype-generic map

- A **generalized map** can be defined by using catamorphism:

$$\mathbf{map}_F\ f = (\!|\, \mathbf{in}_F \circ F(f, \mathsf{id}) \,|\!)$$

  - A bifunctor F decides the shape of trees

- **Providing datatype-generic map fusion law**

$$\mathbf{map}_F\ g \circ \mathbf{map}_F\ f = \mathbf{map}_F\ (g \circ f)$$

Datatype-generic map

- A **generalized map** can be defined by using catamorphism:

$$\mathbf{map}_F\ f = (\!|\ \mathbf{in}_F \circ F(f, \mathrm{id})\ |\!)$$

  - A bifunctor F decides the shape of trees

- **Providing datatype-generic map fusion law**

$$\mathbf{map}_F\ g \circ \mathbf{map}_F\ f = \mathbf{map}_F\ (g \circ f)$$

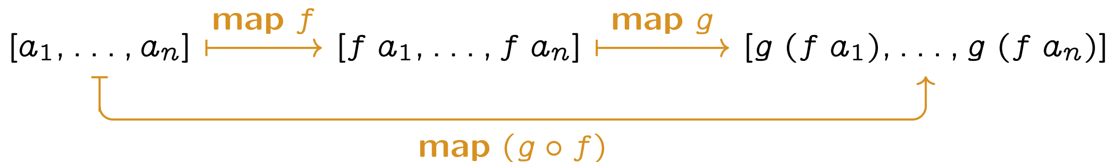
Recursion schemes are a good framework for datatype-generic calculation! 😊

An example of calculation rules

Map fusion law (again)

- famous calculation rule for list functions:

$$(\mathbf{map}\ g) \circ (\mathbf{map}\ f) = \mathbf{map}\ (g \circ f)$$



Datatype-generic rules?

Map fusion law (generalized)

- **Let's generalize to arbitrary datatypes**!

$$(\mathbf{map}_T\ g) \circ (\mathbf{map}_T\ f) = \mathbf{map}_T\ (g \circ f)$$

  - parametrized by (bi)functor T

For Lists: $T(A, X) = \mathbf{1} + A \times X$

$$[a_1, \ldots, a_n] \xmapsto{\mathbf{map}_T\ f} [f\ a_1, \ldots, f\ a_n] \xmapsto{\mathbf{map}_T\ g} [g\ (f\ a_1), \ldots, g\ (f\ a_n)]$$

$\mathbf{map}_T\ (g \circ f)$

Datatype-generic rules?

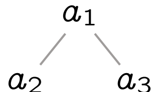
Map fusion law (generalized)

- **Let's generalize to arbitrary datatypes!**

$$(\mathbf{map}_T\ g) \circ (\mathbf{map}_T\ f) = \mathbf{map}_T\ (g \circ f)$$

  - parametrized by (bi)functor T

For Trees: $T(A, X) = \mathbf{1} + A \times X \times A$



Datatype-generic rules?

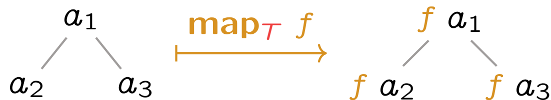
Map fusion law (generalized)

- **Let's generalize to arbitrary datatypes!**

$$(\mathbf{map}_T\ g) \circ (\mathbf{map}_T\ f) = \mathbf{map}_T\ (g \circ f)$$

  - parametrized by (bi)functor T

For Trees: $T(A, X) = \mathbf{1} + A \times X \times A$



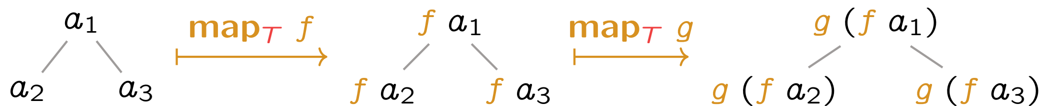
Datatype-generic rules?
Map fusion law (generalized)

- **Let's generalize to arbitrary datatypes!**

$$(\mathbf{map}_T\ g) \circ (\mathbf{map}_T\ f) = \mathbf{map}_T\ (g \circ f)$$

  - parametrized by (bi)functor T

For Trees: $T(A, X) = \mathbf{1} + A \times X \times A$



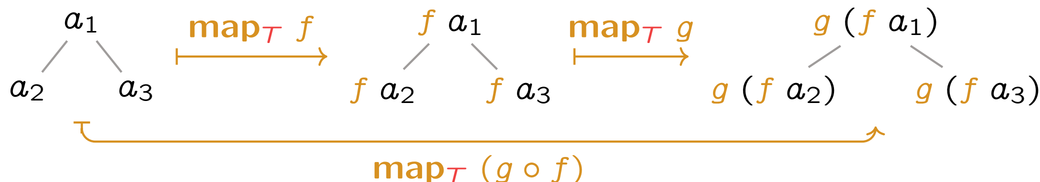
Datatype-generic rules?
Map fusion law (generalized)

- **Let's generalize to arbitrary datatypes!**

$$(\mathbf{map}_T\ g) \circ (\mathbf{map}_T\ f) = \mathbf{map}_T\ (g \circ f)$$

  - parametrized by (bi)functor T

For Trees: $T(A, X) = \mathbf{1} + A \times X \times A$



Power and Limit of Catamorphism

- Catamorphism provides powerful way to define functions:

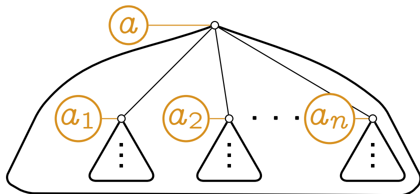
$$\begin{aligned} \textbf{plus}\ x &= (\![\lambda y \Rightarrow x, S]\!) \\ \textbf{times}\ x &= (\![\lambda y \Rightarrow 0, \textbf{plus}\ x]\!) \end{aligned}$$

- But catamorphism **cannot** capture the following function:

$$\begin{aligned} \textbf{fib}\ 0 &= 1 \\ \textbf{fib}\ 1 &= 1 \\ \textbf{fib}\ (n+2) &= \textbf{fib}\ (n+1) + \textbf{fib}\ n \end{aligned}$$

Histomorphism[Uustalu+,99]

- extension of catamorphism
  - capturing Course-of-value iteration
  - can access the **HISTOry of the computation**
- remember and reuse the results of computation for partial trees like **DP**!
- using valued trees (cofree comonad) for memoizing



Definition and application

- the def. of histomorphism $\{\!|\varphi|\!\}_F$

  $f = \varphi \circ F(⟦\langle f, \mathbf{in}_F^{-1} \rangle⟧_{F_C^\times}) \Leftrightarrow f = \{\!|\varphi|\!\}_F$

Definition and application

- the def. of histomorphism $\{\!|\varphi|\!\}_F$

  $f = \varphi \circ F(⟦\langle f, \mathbf{in}_F^{-1} \rangle⟧_{F_C^\times}) \Leftrightarrow f = \{\!|\varphi|\!\}_F$

  This anamorphism builds up memoization tree!

Definition and application

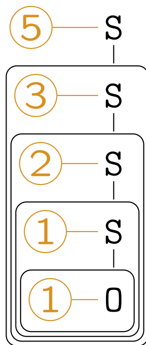
- the def. of histomorphism $\{\!|\varphi|\!\}_F$

$$f = \varphi \circ F([\![\langle f, \mathbf{in}_F^{-1}\rangle]\!]_{F_C^\times}) \Leftrightarrow f = \{\!|\varphi|\!\}_F$$

This anamorphism builds up memoization tree!

- histomorphic definition of fib

$$\begin{aligned} \mathbf{fib} &:= \{\!|[one, f \circ distl \circ out]|\!\} \\ \mathbf{where}\quad f &= [one \circ snd, a] \\ a &= add \circ (id \times fst \circ out) \end{aligned}$$



Outline

1. Introduction
2. Recursion Schemes
3. Coq library
4. Conclusion

naive RS-program
on an datatype
(Runnable)

faster RS-program
on an datatype
(Runnable)

datatype-generic
calculation rules

naive RS-program
on an datatype
(Runnable)

faster RS-program
on an datatype
(Runnable)

datatype-generic
calculation rules

apply

naive RS-program
on an datatype
(Runnable)

faster RS-program
on an datatype
(Runnable)

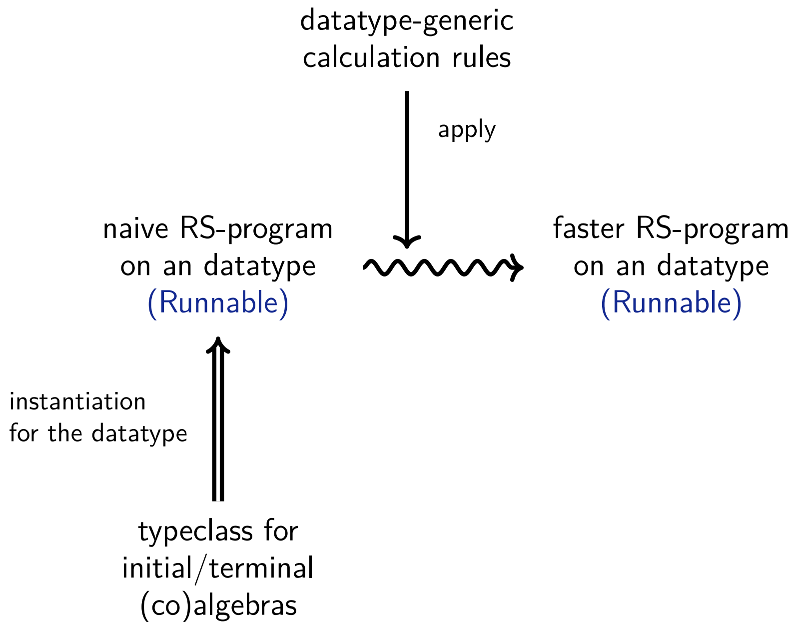
datatype-generic
calculation rules

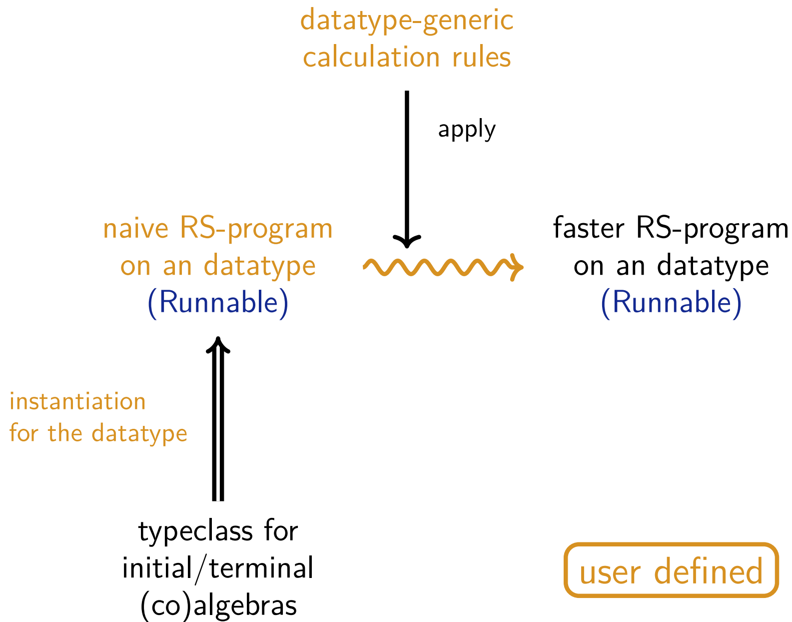
apply

naive RS-program
on an datatype
(Runnable)

faster RS-program
on an datatype
(Runnable)

typeclass for
initial/terminal
(co)algebras


datatype-generic
calculation rules
apply
naive RS-program
on an datatype
(Runnable)
faster RS-program
on an datatype
(Runnable)
instantiation
for the datatype
typeclass for
initial/terminal
(co)algebras


datatype-generic
calculation rules
apply
naive RS-program
on an datatype
(Runnable)
faster RS-program
on an datatype
(Runnable)
instantiation
for the datatype
typeclass for
initial/terminal
(co)algebras
user defined

```
Theorem data_functor :
  forall {A B C : Type} (f : A -> B) (g : B -> C),
    (fmap g) ∘ (fmap f) = fmap (g ∘ f).
Proof.
  intros; unfold fmap.
  assert ( (fmap g) ∘ in_ ∘ F⟨f⟩[id] = in_ ∘ F⟨g ∘ f⟩[id] ∘ F⟨id⟩[fmap g] ).
  {
    unfold fmap.
      Left
    = ( in_ ∘ (F⟨g⟩[id] ∘ F⟨id⟩[⦇in_ ∘ F⟨g⟩[id]⦈]) ∘ F⟨f⟩[id] )
        { by cata_cancel }.
    = ( in_ ∘ (F⟨g⟩[⦇in_ ∘ F⟨g⟩[id]⦈] ∘ F⟨f⟩[id]) )
        { by fmap_functor_dist }.
    = ( in_ ∘ (F⟨g ∘ f⟩[⦇in_ ∘ F⟨g⟩[id]⦈]) )    { by fmap_functor_dist }.
    = ( in_ ∘ (F⟨g ∘ f⟩[id] ∘ F⟨id⟩[⦇in_ ∘ F⟨g⟩[id]⦈]) )
        { by fmap_functor_dist }.
    = Right.
  }
  unfold fmap in H0.
  Left
  = ( ⦇ in_ ∘ F⟨g⟩[id] ⦈ ∘ ⦇ in_ ∘ F⟨f⟩[id] ⦈ ).
  = ( ⦇in_ ∘ F ⟨g ∘ f⟩[id] ⦈ )  { apply cata_fusion }.
  = Right.
Qed.
```

Evaluation: Application

- We formalize all 35 theorems in [Uustalu+ 1999]

Evaluation: Application

- We formalize all 35 theorems in [Uustalu+ 1999]
  - Including histomorphism
  - About 950 lines script

Evaluation: Application

- We formalize all 35 theorems in [Uustalu+ 1999]
  - Including histomorphism
  - About 950 lines script
- enable us to define histomorphic definition

Evaluation: Application

- We formalize all 35 theorems in [Uustalu+ 1999]
  - Including histomorphism
  - About 950 lines script
- enable us to define histomorphic definition
  - Fib, Unbounded Knapsack Problem

Evaluation: Application

- We formalize all 35 theorems in [Uustalu+ 1999]
  - Including histomorphism
  - About 950 lines script
- enable us to define histomorphic definition
  - Fib, Unbounded Knapsack Problem

```
Definition ukp (wvs : list (nat * nat))
:= {| [fun _ => 0,
      fun t => maximize (fun p =>
          match p with
          | (w,v) => match t !! (w - 1) with
                         | Nothing => 0  | Just a  => v + a end) end
      wvs] |}.
```

Definition of (bi)functor (1/3)

- Types for polynomial (bi)functors: `PolyF : Type`

```
Inductive PolyF : Type :=
| zer  : PolyF                     (* empty type *)
| one  : PolyF                     (* unit type *)
| arg1 : PolyF                     (* projection for first arguments *)
| arg2 : PolyF                     (* projection for second arguments *)
| Sum  : PolyF -> PolyF -> PolyF   (* sum types *)
| Prod : PolyF -> PolyF -> PolyF   (* product types *).
```

  - AST of polynomials
    - This is bifunctor: first arguments are used for the index variable (e.g. variable A in the functor $L_A(X) = \mathbf{1} + A \times X$)

Definition of (bi)functor (2/3)

- AST F : `PolyF` to concrete functor $[\![F]\!]\ A : \mathbf{Type} \to \mathbf{Type}$

```
Fixpoint inst (F : PolyF) (A X : Type) : Type :=
  match F with
    zer => Empty_set | one => unit
  | arg1 => A | arg2 => X
  | Sum F G  => (inst F A X) + (inst G A X)
  | Prod F G => (inst F A X) * (inst G A X)
  end.
Notation "[[ F ]]" := (inst F) (at level 20).
```

Definition of (bi)functor (3/3)

- The definition of “fmap”

```
Fixpoint fmap (F : PolyF) {A0 A1 X0 X1 : Type}
  (f : A0 -> A1) (g : X0 -> X1) : ⟦F⟧ A0 X0 -> ⟦F⟧ A1 X1 :=
  match F with
    zer => id | one => id | arg1 => f | arg2 => g
  | Sum F G  => fun x
              => match x with
                   | inl x => inl (fmap F f g x)
                   | inr x => inr (fmap G f g x)
                   end
  | Prod F G => fun x
              => (fmap F f g (fst x), fmap G f g (snd x))
  end.
```

- proof of functor laws for $\llbracket F \rrbracket\ A : \mathbf{Type} \to \mathbf{Type}$
- other notations

Typeclass for initial algebra

```
Class F_initial_algebra (F : PolyF) (A mF : Type)
:={
    cata : forall (X : Type), (⟦F⟧ A X -> X) -> (mF -> X);
    in_  : ⟦F⟧ A mF -> mF;
    cata_charn : forall (X : Type) (f : mF -> X) (φ : ⟦F⟧ A X -> X),
        f ∘ in_ = φ ∘ F[f] <-> f = cata X φ
  }.
```

- simply formalization of initial algebras and catamorphism
  - instances of this typeclass are Coq program for $\mathbf{in}_F$ and $(\!|\, \varphi \,|\!)_F$
  - the proof of Cata-Charn is also required

Outline

1. Introduction
2. Recursion Schemes
3. Coq library
4. Conclusion

Our Contribution

- A Coq library for **datatype-generic** program calculation
  - based on [Tesson+, 2011]
  - This library enable users to write the datatype-generic proof in the **equational style**
- We give Coq proofs of all theorems in [Uustalu+, 1999]
  - Their work describes formal definitions and calculation rules for various kinds of recursion schemes in category **Set**
  - They propose histomorphisms and futumorphisms

Folding

(tearing down the inductive datatypes)

Catamorphism[Malcom, 1990]

$$f = \varphi \circ F(f) \circ \mathbf{in}_F^{-1}$$

Paramorphism[Meertens, 1992]

$$f = \varphi \circ F(\langle f, \mathrm{id}_{\mu F} \rangle) \circ \mathbf{in}_F^{-1}$$

Histomorphism[Uustalu+, 1999]

$$f = \varphi \circ F([\![\langle f, \mathbf{in}_F^{-1} \rangle]\!]) \circ \mathbf{in}_F^{-1}$$

Unfolding

(building up the coinductive datatypes)

Anamorphism[Fokkinga+, 1991]

$$f = \mathbf{out}_F^{-1} \circ F(f) \circ \psi$$

Apomorphism[Vos, 1995]

$$f = \mathbf{out}_F^{-1} \circ F([f, \mathrm{id}_{\nu F}]) \circ \psi$$

Futumorphism[Uustalu+, 1999]

$$f = \mathbf{out}_F^{-1} \circ F((\!| [f, \mathbf{out}_F^{-1}] |\!)) \circ \psi$$

Refolding

(building up an intermediate data structure then tearing down it)

Hylomorphism[Meijer+, 1991]

cata ∘ *ana*

Dynamorphism[Kabanov+, 2006]

histo ∘ *ana*

Future Work: hylo- and dyna- morphisms

- formalizing hylomorphisms and dynamorphisms
  - important for formalizing devide-and-canquer algorithm and dynamic-programming algorithm
  - only defined in algebraically compact category such as $\mathbf{Cpo}_{\perp}$
- but in Coq, inductive datatypes and coinductive datatypes do **not coincide**.
- how formalize?