Verification of Programs with Truly Nested Datatypes in Coq

Ralph Matthes

Institut de Recherche en Informatique de Toulouse (IRIT), CNRS
Équipe ACADIE
(Assistance à la Certification de Systèmes Distribués et Embarqués)

Conference of the Types Project – TYPES 2006
University of Nottingham, U. K.
April 20, 2006

Abstract

Truly nested datatypes are families of datatypes that are indexed over all types such that the constructors relate different family members (unlike the homogeneous lists). Moreover, even the family name is involved in the expression that gives the index the argument type of the constructor refers to. Although these families are not directly definable in Coq, there are systems of rewrite rules that are known to yield only terminating functions and that provide the means of iterating over elements of these families. However, there do not seem to exist genuine induction principles (the dependently-typed versions of recursors and not just iterators). A way out (that is surprisingly close to the TYPES '03 talk of P. Aczel) will be shown and illustrated with examples that have been fully formalized in Coq.

Outline

1. Nested Datatypes
   - Heterogeneous Datatypes
   - Truly Nested Datatypes
2. Verification
   - Induction for Mendler's Style
   - More Properties for the Approximation
   - Solution with Inductive-Recursive Definitions

Outline

1. Nested Datatypes
   - Heterogeneous Datatypes
   - Truly Nested Datatypes
2. Verification
   - Induction for Mendler's Style
   - More Properties for the Approximation
   - Solution with Inductive-Recursive Definitions

Relating different family members

```
Coq <   Inductive List (A:Set) : Set :=
Coq <   | nil : List A
Coq <   | cons : A -> List A -> List A.
List is defined
List_rect is defined
List_ind is defined
List_rec is defined

Coq <   Fixpoint map (A B:Set)(f:A->B)(l:List A){struct l}
Coq <     :List B :=
Coq <     match l with
Coq <           |      nil => nil _
Coq <           | cons a l'=> cons (f a) (map f l') end.

Coq < Check map.
map
     : forall A B : Set, (A -> B) -> List A -> List B
```

ACADIE

Relating different family members

```
Coq <   Inductive List (A:Set) : Set :=
Coq <   | nil : List A
Coq <   | cons : A -> List A -> List A.
List is defined
List_rect is defined
List_ind is defined
List_rec is defined

Coq <   Fixpoint map (A B:Set)(f:A->B)(l:List A){struct l}
Coq <     :List B :=
Coq <     match l with
Coq <           |      nil => nil _
Coq <           | cons a l'=> cons (f a) (map f l') end.

Coq < Check map.
map
     : forall A B : Set, (A -> B) -> List A -> List B
```

ACADIE

Outline

1. Nested Datatypes
   - Heterogeneous Datatypes
   - Truly Nested Datatypes
2. Verification
   - Induction for Mendler's Style
   - More Properties for the Approximation
   - Solution with Inductive-Recursive Definitions

powerlists

```
Coq < Inductive PList : Set->Type:=
Coq <   | zero : forall A:Set, A -> PList A
Coq <   | succ : forall A:Set, PList (A * A)%type -> PList A.
```

Note the target type `Type` which is required with predicative `Set`.

```
Coq < Definition myPList : PList nat :=
Coq <  succ (succ (succ (zero (((1,2),(3,4)),((5,6),(7,8)))))).
```

Clearly, every inhabitant of `PList A` contains 2^n elements of A for some (unspecified) n.

powerlists

```
Coq < Inductive PList : Set->Type:=
Coq <   | zero : forall A:Set, A -> PList A
Coq <   | succ : forall A:Set, PList (A * A)%type -> PList A.
```

Note the target type `Type` which is required with predicative `Set`.

```
Coq < Definition myPList : PList nat :=
Coq <  succ (succ (succ (zero (((1,2),(3,4)),((5,6),(7,8)))))).
```

Clearly, every inhabitant of `PList A` contains 2^n elements of A for some (unspecified) n.

powerlists

```
Coq < Inductive PList : Set->Type:=
Coq <   | zero : forall A:Set, A -> PList A
Coq <   | succ : forall A:Set, PList (A * A)%type -> PList A.
```

Note the target type `Type` which is required with predicative `Set`.

```
Coq < Definition myPList : PList nat :=
Coq <  succ (succ (succ (zero (((1,2),(3,4)),((5,6),(7,8)))))).
```

Clearly, every inhabitant of `PList A` contains 2^n elements of A for some (unspecified) n.

powerlists

```
Coq < Inductive PList : Set->Type:=
Coq <   | zero : forall A:Set, A -> PList A
Coq <   | succ : forall A:Set, PList (A * A)%type -> PList A.
```

Note the target type `Type` which is required with predicative `Set`.

```
Coq < Definition myPList : PList nat :=
Coq <  succ (succ (succ (zero (((1,2),(3,4)),((5,6),(7,8)))))).
```

Clearly, every inhabitant of `PList A` contains 2^n elements of A for some (unspecified) n.

powerlists into lists

```
Coq < Fixpoint unzip (A:Set)(l:List (A*A)){struct l}:List A:=
Coq <  match l return List A with
Coq <    | nil                => nil _
Coq <    | cons (a1,a2) l' => cons a1 (cons a2 (unzip l'))
Coq <  end.

Coq < Fixpoint PListToList(A:Set)(l:PList A){struct l}:List A:=
Coq <  match l in PList A return List A with
Coq <     | zero _ a  => cons a (nil _)
Coq <     | succ _ l' => unzip (PListToList l')
Coq <  end.
```

`Eval compute in (PListToList myPList).` gives the list of elements $1, \ldots, 8$.

powerlists into lists

```
Coq < Fixpoint unzip (A:Set)(l:List (A*A)){struct l}:List A:=
Coq <  match l return List A with
Coq <    | nil                => nil _
Coq <    | cons (a1,a2) l' => cons a1 (cons a2 (unzip l'))
Coq <  end.

Coq < Fixpoint PListToList(A:Set)(l:PList A){struct l}:List A:=
Coq <  match l in PList A return List A with
Coq <     | zero _ a  => cons a (nil _)
Coq <     | succ _ l' => unzip (PListToList l')
Coq <  end.
```

`Eval compute in (PListToList myPList).` gives the list of elements $1, \ldots, 8$.

programming is more demanding: summing up a powerlist

We want a function `sumPList : PList nat -> nat` that sums up the elements. First specify:

```
Coq <   Fixpoint sumList (l:List nat) : nat :=
Coq <   match l with nil => 0
Coq <   | cons n l' => n + sumList l' end.

Coq <   Definition sumPListSpec := forall (l:PList nat),
Coq <     sumPList l = sumList (PListToList l).
```

This is an algorithm, but we may do deforestation to it.
Problem:

`sumPList (succ l) :=??` for `l:PList(nat*nat)`.

programming is more demanding: summing up a powerlist

We want a function `sumPList : PList nat -> nat` that sums up the elements. First specify:

```
Coq <   Fixpoint sumList (l:List nat) : nat :=
Coq <   match l with nil => 0
Coq <   | cons n l' => n + sumList l' end.

Coq <   Definition sumPListSpec := forall (l:PList nat),
Coq <     sumPList l = sumList (PListToList l).
```

This is an algorithm, but we may do deforestation to it.
Problem:

`sumPList (succ l) :=??` for `l:PList(nat*nat)`.

solution

Define a more general function:

```
Coq < Fixpoint sumPList' (A:Set)(l:PList A){struct l}
Coq <  : (A->nat)->nat :=
Coq <  match l in PList A return (A->nat)->nat with
Coq <  | zero A a => fun f => f a
Coq <  | succ _ l' => fun f => sumPList' l'
Coq <     (fun a => let(a1,a2):=a in f a1 + f a2) end.

Coq < Definition sumPList l := sumPList' l (fun x=>x).
```

A more general specification:

```
Coq < Definition sumPList'Spec :=
Coq <  forall (A:Set)(l:PList A)(f:A->nat),
Coq <   sumPList' l f = sumList(map f (PListToList l)).
```

ACADIE

solution

Define a more general function:

```
Coq < Fixpoint sumPList' (A:Set)(l:PList A){struct l}
Coq <  : (A->nat)->nat :=
Coq <  match l in PList A return (A->nat)->nat with
Coq <  | zero A a => fun f => f a
Coq <  | succ _ l' => fun f => sumPList' l'
Coq <     (fun a => let(a1,a2):=a in f a1 + f a2) end.

Coq < Definition sumPList l := sumPList' l (fun x=>x).
```

A more general specification:

```
Coq < Definition sumPList'Spec :=
Coq <  forall (A:Set)(l:PList A)(f:A->nat),
Coq <   sumPList' l f = sumList(map f (PListToList l)).
```

ACADIE

proving the specification

Coq automatically generates an induction principle for `PList`:

```
Coq < Check PList_ind :
Coq <   forall P : (forall A : Set, PList A -> Prop),
Coq <     (forall (A : Set) (a : A), P A (zero a)) ->
Coq <     (forall (A : Set) (l : PList (A * A)),
Coq <       P (A * A)%type l -> P A (succ l)) ->
Coq <     forall (A : Set) (l : PList A), P A l.
```

This can still be understood by induction with a measure.
The specification before can be readily verified with this principle.

proving the specification

Coq automatically generates an induction principle for `PList`:

```
Coq < Check PList_ind :
Coq <   forall P : (forall A : Set, PList A -> Prop),
Coq <     (forall (A : Set) (a : A), P A (zero a)) ->
Coq <     (forall (A : Set) (l : PList (A * A)),
Coq <       P (A * A)%type l -> P A (succ l)) ->
Coq <     forall (A : Set) (l : PList A), P A l.
```

This can still be understood by induction with a measure.

The specification before can be readily verified with this principle.

proving the specification

Coq automatically generates an induction principle for `PList`:

```
Coq < Check PList_ind :
Coq <   forall P : (forall A : Set, PList A -> Prop),
Coq <     (forall (A : Set) (a : A), P A (zero a)) ->
Coq <     (forall (A : Set) (l : PList (A * A)),
Coq <       P (A * A)%type l -> P A (succ l)) ->
Coq <     forall (A : Set) (l : PList A), P A l.
```

This can still be understood by induction with a measure.
The specification before can be readily verified with this principle.

Outline

1. Nested Datatypes
   - Heterogeneous Datatypes
   - Truly Nested Datatypes
2. Verification
   - Induction for Mendler's Style
   - More Properties for the Approximation
   - Solution with Inductive-Recursive Definitions

3-bushes

The obvious "complexification" of the 2-bushes by Bird et al.
In Coq, they can be introduced axiomatically:

```
Coq < Variable Bsh3 : Set->Set.

Coq < Variable bnil3 : forall (A:Set), Bsh3 A.

Coq < Variable bcons3: forall (A:Set),
Coq <   A -> Bsh3(Bsh3(Bsh3 A)) -> Bsh3 A.
```

The second argument of `bcons3` is more complex than `Bsh3 A`.
With powerlists, the family index has been `A * A`.
Here, it is `Bsh3(Bsh3 A)`.
A reference to the family itself. And even a nested reference.
This second nesting only adds spice but is not required
to qualify as truly nested.

3-bushes

The obvious "complexification" of the 2-bushes by Bird et al.
In Coq, they can be introduced axiomatically:

```
Coq < Variable Bsh3 : Set->Set.

Coq < Variable bnil3 : forall (A:Set), Bsh3 A.

Coq < Variable bcons3: forall (A:Set),
Coq <   A -> Bsh3(Bsh3(Bsh3 A)) -> Bsh3 A.
```

The second argument of `bcons3` is more complex than `Bsh3 A`.
With powerlists, the family index has been `A * A`.
Here, it is `Bsh3(Bsh3 A)`.
A reference to the family itself. And even a nested reference.
This second nesting only adds spice but is not required
to qualify as truly nested.

3-bushes

The obvious "complexification" of the 2-bushes by Bird et al.
In Coq, they can be introduced axiomatically:

```
Coq < Variable Bsh3 : Set->Set.

Coq < Variable bnil3 : forall (A:Set), Bsh3 A.

Coq < Variable bcons3: forall (A:Set),
Coq <   A -> Bsh3(Bsh3(Bsh3 A)) -> Bsh3 A.
```

The second argument of `bcons3` is more complex than `Bsh3 A`.
With powerlists, the family index has been `A * A`.
Here, it is `Bsh3(Bsh3 A)`.
A reference to the family itself. And even a nested reference.
This second nesting only adds spice but is not required
to qualify as truly nested.

Definition

A *truly nested datatype* is a nested datatype with a call to the family name within a type argument of an argument of one of the datatype constructors.

```
bcons3:  A -> Bsh3(Bsh3(Bsh3 A)) -> Bsh3 A
```

enumerating ternary trees

```
Coq < Inductive Tri : Set := L:Tri | N:Tri->Tri->Tri->Tri.
```

We want to have the 3-bush with all the ternary trees of height less than m.
For $m := 3$, this would be the following list:

```
Coq < Definition myTriList :=  L :: N L L L
Coq <  :: N (N L L L) L L
Coq <    :: N L (N L L L) L
Coq <      :: N (N L L L) (N L L L) L
Coq <        :: N L L (N L L L)
Coq <          :: N (N L L L) L (N L L L)
Coq <            :: N L (N L L L) (N L L L)
Coq <              :: N (N L L L) (N L L L) (N L L L)
Coq <                :: nil.
```

enumerating ternary trees

```
Coq < Inductive Tri : Set := L:Tri | N:Tri->Tri->Tri->Tri.
```

We want to have the 3-bush with all the ternary trees of height less than m.

For $m := 3$, this would be the following list:

```
Coq < Definition myTriList :=  L :: N L L L
Coq <  :: N (N L L L) L L
Coq <    :: N L (N L L L) L
Coq <      :: N (N L L L) (N L L L) L
Coq <        :: N L L (N L L L)
Coq <          :: N (N L L L) L (N L L L)
Coq <            :: N L (N L L L) (N L L L)
Coq <              :: N (N L L L) (N L L L) (N L L L)
Coq <                :: nil.
```

enumerating ternary trees

```
Coq < Inductive Tri : Set := L:Tri | N:Tri->Tri->Tri->Tri.
```

We want to have the 3-bush with all the ternary trees of height less than m.
For $m := 3$, this would be the following list:

```
Coq < Definition myTriList :=  L :: N L L L
Coq <  :: N (N L L L) L L
Coq <    :: N L (N L L L) L
Coq <      :: N (N L L L) (N L L L) L
Coq <        :: N L L (N L L L)
Coq <          :: N (N L L L) L (N L L L)
Coq <            :: N L (N L L L) (N L L L)
Coq <              :: N (N L L L) (N L L L) (N L L L)
Coq <                :: nil.
```

```
Coq < Definition mkTriBsh3' : nat ->
Coq <                  forall (A:Set), (Tri->A) -> Bsh3 A :=
Coq < fun m A f =>
Coq <  (fix F (n:nat)(A:Set)(f:Tri->A){struct n} : Bsh3 A :=
Coq <    match n with
Coq <    |   0 => bnil3 _
Coq <    | S m => bcons3 (f L) (F m _ (fun t1 => F m _
Coq <             (fun t2 => (F m _ (fun t3 => f(N t3 t2 t1)))))
Coq <     end)
Coq <   m A f.

Coq < Definition mkTriBsh3 (m:nat) := mkTriBsh3' m (fun x=>x).
```

mkTriBsh3 3 has the 9 desired elements, but contains 29 bcons3 and 21 bnil3. mkTriBsh3 4 has the 730 desired elements and 2640 bcons3 and 1911 bnil3.

```
Coq < Definition mkTriBsh3' : nat ->
Coq <                  forall (A:Set), (Tri->A) -> Bsh3 A :=
Coq < fun m A f =>
Coq <  (fix F (n:nat)(A:Set)(f:Tri->A){struct n} : Bsh3 A :=
Coq <    match n with
Coq <    |   0 => bnil3 _
Coq <    | S m => bcons3 (f L) (F m _ (fun t1 => F m _
Coq <             (fun t2 => (F m _ (fun t3 => f(N t3 t2 t1)))))
Coq <     end)
Coq <   m A f.

Coq < Definition mkTriBsh3 (m:nat) := mkTriBsh3' m (fun x=>x).
```

mkTriBsh3 3 has the 9 desired elements, but contains 29 bcons3 and 21 bnil3. mkTriBsh3 4 has the 730 desired elements and 2640 bcons3 and 1911 bnil3.

the question

How can we analyze the elements of `Bsh3 Tri`?

Currently, we only have the constructors.
And they are not accepted by Coq for an inductive definition ("not strictly positive").

The computational solution appears in a joint paper with Andreas Abel and Tarmo Uustalu (TCS 333(1-2), pp. 3-66, 2005).
We propose inductive type families with an iterator in the style of Nax Mendler that guarantees termination of all functions expressible in this (definitional) extension of system F^ω.

But there has been no logical system that allows to reason by induction on the structure of those inductive families.
This is the new contribution of the present talk.

the question

How can we analyze the elements of `Bsh3 Tri`?

Currently, we only have the constructors.
And they are not accepted by Coq for an inductive definition ("not strictly positive").

The computational solution appears in a joint paper with Andreas Abel and Tarmo Uustalu (TCS 333(1-2), pp. 3-66, 2005).
We propose inductive type families with an iterator in the style of Nax Mendler that guarantees termination of all functions expressible in this (definitional) extension of system F^ω.

But there has been no logical system that allows to reason by induction on the structure of those inductive families.
This is the new contribution of the present talk.

the question

How can we analyze the elements of `Bsh3 Tri`?

Currently, we only have the constructors.
And they are not accepted by Coq for an inductive definition ("not strictly positive").

The computational solution appears in a joint paper with Andreas Abel and Tarmo Uustalu (TCS 333(1-2), pp. 3-66, 2005).
We propose inductive type families with an iterator in the style of Nax Mendler that guarantees termination of all functions expressible in this (definitional) extension of system F^ω.

But there has been no logical system that allows to reason by induction on the structure of those inductive families.
This is the new contribution of the present talk.

Outline

1. Nested Datatypes
   - Heterogeneous Datatypes
   - Truly Nested Datatypes
2. Verification
   - Induction for Mendler's Style
   - More Properties for the Approximation
   - Solution with Inductive-Recursive Definitions

```
Definition k0 := Set.
Definition k1 := k0 -> k0.
Definition k2 := k1 -> k1.
Definition sub_k1 (X Y:k1) : Set := forall A:Set, X A -> Y A.
Infix "c_k1" := sub_k1 (at level 60).

Variable F : k2.
Variable mu2 : k1
Variable InStd : F mu2 c_k1 mu2
Variable MIt: forall G : k1,
       (forall X : k1, X c_k1 G -> F X c_k1 G) -> mu2 c_k1 G.
Assumption MItRedStd : forall (G:k1)
   (s:forall X:k1, X c_k1 G -> F X c_k1 G)(A:Set)(t:F mu2 A),
       MIt s (InStd t) = s _ (MIt s) _ t.
```

What would be an induction principle for `mu2`?
It should be a dependently-typed version of `MIt`.

generalization of the constructor

Idea: generalize the system of Uustalu and Vene to rank 2.

$$In : \forall X : \kappa 1.\, X \subseteq \mu_2 \to FX \subseteq \mu_2$$

instead of

$$InStd : F\mu_2 \subseteq \mu_2$$

And this is even accepted as an inductive definition in Coq with *In* the single constructor of μ_2.
The minimality scheme for sort *Set* generated by Coq has type

$$\forall G : \kappa 1.\, (\forall X : \kappa 1.\, X \subseteq \mu_2 \to X \subseteq G \to FX \subseteq G) \to \mu_2 \subseteq G$$

This is even the lifting of the type of Mendler's recursor to nested datatypes.

generalization of the constructor

Idea: generalize the system of Uustalu and Vene to rank 2.

$$In : \forall X : \kappa 1.\, X \subseteq \mu_2 \to FX \subseteq \mu_2$$

instead of

$$InStd : F\mu_2 \subseteq \mu_2$$

And this is even accepted as an inductive definition in Coq with *In* the single constructor of μ_2.

The minimality scheme for sort *Set* generated by Coq has type

$$\forall G : \kappa 1.\, (\forall X : \kappa 1.\, X \subseteq \mu_2 \to X \subseteq G \to FX \subseteq G) \to \mu_2 \subseteq G$$

This is even the lifting of the type of Mendler's recursor to nested datatypes.

generalization of the constructor

Idea: generalize the system of Uustalu and Vene to rank 2.

$$In : \forall X : \kappa 1.\, X \subseteq \mu_2 \to FX \subseteq \mu_2$$

instead of

$$InStd : F\mu_2 \subseteq \mu_2$$

And this is even accepted as an inductive definition in Coq with *In* the single constructor of μ_2.
The minimality scheme for sort *Set* generated by Coq has type

$$\forall G : \kappa 1.\, (\forall X : \kappa 1.\, X \subseteq \mu_2 \to X \subseteq G \to FX \subseteq G) \to \mu_2 \subseteq G$$

This is even the lifting of the type of Mendler's recursor to nested datatypes.

generalization of the constructor

Idea: generalize the system of Uustalu and Vene to rank 2.

$$In : \forall X : \kappa 1.\, X \subseteq \mu_2 \to FX \subseteq \mu_2$$

instead of

$$InStd : F\mu_2 \subseteq \mu_2$$

And this is even accepted as an inductive definition in Coq with *In* the single constructor of μ_2.
The minimality scheme for sort *Set* generated by Coq has type

$$\forall G : \kappa 1.\, (\forall X : \kappa 1.\, X \subseteq \mu_2 \to X \subseteq G \to FX \subseteq G) \to \mu_2 \subseteq G$$

This is even the lifting of the type of Mendler's recursor to nested datatypes.

the generated induction principle

Coq's induction principle supports the following reasoning:
Given a predicate $P : \forall A.\, \mu_2 A \to \mathit{Prop}$, we may deduce P holds universally – $\forall A\, \forall r : \mu_2 A.\, P_A\, r$, if,
for every $X : \kappa 1$ and every $j : X \subseteq \mu_2$, from the *inductive hypothesis*

$$\forall A\, \forall x : XA,\ P_A(j\, A\, x)$$

we can infer (this is called the *inductive step*)

$$\forall A\, \forall t : FXA.\, P_A(\mathit{In}\, j\, t)\ .$$

What does the inductive hypothesis say for $X := \mu_2$ and $j := \lambda A \lambda x : A.x$? $\forall A\, \forall r : \mu_2 A.\, P_A\, r$!!
Hence, only for the non-canonical elements of μ_2 that are not introduced by *InStd*, the inductive step requires some work.
Thus, verification completely rests on those non-canonical elements.

the generated induction principle

Coq's induction principle supports the following reasoning:
Given a predicate $P : \forall A.\, \mu_2 A \to \mathit{Prop}$, we may deduce P holds universally – $\forall A\, \forall r : \mu_2 A.\, P_A\, r$, if,
for every $X : \kappa 1$ and every $j : X \subseteq \mu_2$, from the *inductive hypothesis*

$$\forall A\, \forall x : XA,\ P_A(j\, A\, x)$$

we can infer (this is called the *inductive step*)

$$\forall A\, \forall t : FXA.\, P_A(\mathit{In}\, j\, t)\ .$$

What does the inductive hypothesis say for $X := \mu_2$ and $j := \lambda A \lambda x : A.x$? $\forall A\, \forall r : \mu_2 A.\, P_A\, r$!!
Hence, only for the non-canonical elements of μ_2 that are not introduced by *InStd*, the inductive step requires some work. Thus, verification completely rests on those non-canonical elements.

the generated induction principle

Coq's induction principle supports the following reasoning:
Given a predicate $P : \forall A.\, \mu_2 A \to \mathit{Prop}$, we may deduce P holds universally – $\forall A \forall r : \mu_2 A.\, P_A\, r$, if,
for every $X : \kappa 1$ and every $j : X \subseteq \mu_2$, from the *inductive hypothesis*

$$\forall A \forall x : XA,\ P_A(j\, A\, x)$$

we can infer (this is called the *inductive step*)

$$\forall A \forall t : FXA.\, P_A(\mathit{In}\, j\, t)\ .$$

What does the inductive hypothesis say for $X := \mu_2$ and $j := \lambda A \lambda x : A.x$? $\forall A \forall r : \mu_2 A.\, P_A\, r$!!
Hence, only for the non-canonical elements of μ_2 that are not introduced by *InStd*, the inductive step requires some work.
Thus, verification completely rests on those non-canonical elements.

more on those non-canonical elements

We need to say what the iterative functions do on the non-canonical elements. The iterator that specializes the recursor satisfies the following computational rule – even w. r. t. convertibility in Coq.

```
Definition comp (A B C:Set)(g:B->C)(f:A->B) : A->C
    := fun x => g (f x).
Infix "o" := comp (at level 90).
Lemma MItRed : forall (G : k1)
  (s : forall X : k1, X c_k1 G -> F X c_k1 G)
  (X : k1)(j: X c_k1 mu2)(A:Set)(t:F X A),
   MIt s (In j t) = s X (fun A => (MIt s (A:=A)) o (j A)) A t.
```

Outline

1. Nested Datatypes
   - Heterogeneous Datatypes
   - Truly Nested Datatypes
2. Verification
   - Induction for Mendler's Style
   - More Properties for the Approximation
   - Solution with Inductive-Recursive Definitions

what remains to be done

Everything.

We did not yet prove anything interesting with the present system.
Evidently, we want to support the usual reasoning that profits from structured programming, the "Algebra of Programming".
This means laws inspired from category theory such a functoriality or naturality.
And this does not work at all in the present framework. The step term of `MIt` has to be too generic:

$$\forall X : \kappa 1.\, X \subseteq G \rightarrow FX \subseteq G$$

Note that we had removed the condition $X \subseteq \mu_2$ because we only wanted iteration.

the idea

Instead of that removed condition, we enter new information about the approximation X:

- X is monotone, witnessed by some term m of type $\forall A \forall B.\, (A \to B) \to XA \to XB$.
- m is functorial, i. e., it satisfies the two functor laws.
- m is only dependent on the extension of its functional argument.

The last one is important since rewriting in Coq is not done under binders.

With these, functoriality properties of the map for μ_2 can be established. But not naturality of the iteratively defined polymorphic functions of type $\mu_2 \subseteq G$.

adding naturality

We want to have the following datatype constructor:

```
(** natural transformations from (X,mX) to (Y,mY) *)
Definition NAT(X Y:k1)(j:X c_k1 Y)(mX:mon X)(mY:mon Y):Prop:=
  forall (A B:Set)(f:A->B)(t:X A),
         j B (mX A B f t) = mY _ _ f (j A t).

In : forall (G:k1)(m:mon G), ext m -> fct1 m -> fct2 m ->
     forall j: G c_k1 mu2, NAT j m mapmu2 -> F G c_k1 mu2.
```

What is `mapmu2`? It should be the map function for `mu2`.
But we are about to define `mu2`. So, this cannot be done in Coq.

Outline

1. Nested Datatypes
   - Heterogeneous Datatypes
   - Truly Nested Datatypes
2. Verification
   - Induction for Mendler's Style
   - More Properties for the Approximation
   - Solution with Inductive-Recursive Definitions

Simultaneous inductive-recursive definitions have been proposed by Peter Dybjer. Here, we may use an impredicative version of them. So, μ_2 is an inductively defined family, and simultaneously, the function

$$map\mu_2 : \forall A \forall B.(A \to B) \to \mu_2 A \to \mu_2 B$$

is defined. Its type is isomorphic with

$$\forall A.\, \mu_2 A \to \forall B.(A \to B) \to \mu_2 B$$

This can nevertheless be implemented in Coq by using an unpublished manuscript by Venanzio Capretta.

what remains to be done

Capretta defined a more liberal family and then by an inductive set which are the good elements (to be represented as a `sig` of Coq). What he does not provide, is an induction principle. In my case it is:

```
mu2Ind : Prop :=
  forall P : (forall A : Set, mu2 A -> Prop),
       (forall (X : k1) (m : mon X) (e : ext m) (f1 : fct1 m)
          (f2 : fct2 m) (j : X c_k1 mu2)(n: NAT j m mapmu2),
          (forall (A : Set) (x : X A), P A (j A x)) ->
        forall (A:Set)(t : F X A), P A (In e f1 f2 n t)) ->
    forall (A : Set) (r : mu2 A), P A r.
```

Is has been proved in Coq with proof-irrelevance.

In this extension, one can show uniqueness and naturality of iteratively defined functions. The general results can be instantiated to treat `Bsh3`.

Conclusions

It is now possible to combine the following benefits:

- termination of all functions following the recursion schemes
- recursion schemes are type-based and not syntax-driven
- genericity: no specific shape of the datatype functors required
- no continuity properties required
- includes truly nested datatypes
- categorical laws for program verification
- program execution within the convertibility relation of Coq

To do:

- More examples
- Reasoning principles for conventional style without non-canonical elements
- Primitive recursion (not only iteration)

Thank you for listening.

Conclusions

It is now possible to combine the following benefits:

- termination of all functions following the recursion schemes
- recursion schemes are type-based and not syntax-driven
- genericity: no specific shape of the datatype functors required
- no continuity properties required
- includes truly nested datatypes
- categorical laws for program verification
- program execution within the convertibility relation of Coq

To do:

- More examples
- Reasoning principles for conventional style without non-canonical elements
- Primitive recursion (not only iteration)

Thank you for listening.

Conclusions

It is now possible to combine the following benefits:

- termination of all functions following the recursion schemes
- recursion schemes are type-based and not syntax-driven
- genericity: no specific shape of the datatype functors required
- no continuity properties required
- includes truly nested datatypes
- categorical laws for program verification
- program execution within the convertibility relation of Coq

To do:

- More examples
- Reasoning principles for conventional style without non-canonical elements
- Primitive recursion (not only iteration)

Thank you for listening.

ACADIE