Witnessing (Co)Datatypes

A Proof Assistant Perspective

Jasmin Blanchette, Andrei Popescu, Dmitriy Traytel

Inria Nancy & LORIA / MPI Saarbrücken
Middlesex University
Technical University Munich

Overview

- Proof Assistants versus Programming Languages
- The Higher Responsibility of Proof Assistants
- Our Work on Isabelle/HOL

Proof Assistants versus Programming Languages

Programming Language (PL)

Proof Assistants versus Programming Languages

Programming Language (PL)

Proof Assistant (PA)

Proof Assistants versus Programming Languages

Programming Language (PL)

Automatic Code Generation

Proof Assistant (PA)

Proof Assistants versus Programming Languages

Programming Language (PL)

Automatic Code Generation

Proof Assistant (PA)

PA = Smart PL

Proof Assistants are Smart

```
fact : nat -> nat
fact n = case n of 0 => 1
                 | Suc m => n * fact m
```

Proof Assistants are Smart

```
fact : nat -> nat
fact n = case n of 0 => 1
                 |  Suc m => n * fact m
```

For a PL, this is just a partial function

Proof Assistants are Smart

```
fact : nat -> nat
fact n = case n of 0 => 1
                 | Suc m => n * fact m
```

For a PL, this is just a partial function

A PA also knows that it terminates

Proof Assistants are Smart

```
fact : nat -> nat
fact n = case n of 0 => 1
                 |  Suc m => n * fact m
```

For a PL, this is just a partial function
A PA also knows that it terminates

```
+ : nat stream -> nat stream -> nat stream
```

Proof Assistants are Smart

```
fact : nat -> nat
fact n = case n of 0 => 1
                 | Suc m => n * fact m
```

For a PL, this is just a partial function
A PA also knows that it terminates

```
+ : nat stream -> nat stream -> nat stream
```

```
fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

Proof Assistants are Smart

```
fact : nat -> nat
fact n = case n of 0 => 1
                 |  Suc m => n * fact m
```

For a PL, this is just a partial function
A PA also knows that it terminates

```
+ : nat stream -> nat stream -> nat stream
```

```
fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

Haskell of course accepts this.

Proof Assistants are Smart

```
fact : nat -> nat
fact n = case n of 0 => 1
                 |  Suc m => n * fact m
```

For a PL, this is just a partial function
A PA also knows that it terminates

```
+ : nat stream -> nat stream -> nat stream
```

```
fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

Haskell of course accepts this. But is it productive?

Proof Assistants are Smart

```
fact : nat -> nat
fact n = case n of 0 => 1
                 |  Suc m => n * fact m
```

For a PL, this is just a partial function
A PA also knows that it terminates

```
+ : nat stream -> nat stream -> nat stream
```

```
fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

Haskell of course accepts this. But is it productive?
A PA would have to understand why this is productive in order to accept it!

The Higher Responsibility of Proof Assistants

```
inc : nat stream -> nat stream
```

The Higher Responsibility of Proof Assistants

```
inc : nat stream -> nat stream
```

Example: inc [0,1,2,...] $=$ [1,2,3,...]

The Higher Responsibility of Proof Assistants

```
inc : nat stream -> nat stream
```

Example: inc [0,1,2,...] = [1,2,3,...]

```
nasty : nat -> nat stream
nasty n = if n < 2
          then Cons n (nasty (n+1))
          else inc (tail (nasty n))
```

The Higher Responsibility of Proof Assistants

```
inc : nat stream -> nat stream
```

Example: inc [0,1,2,...] = [1,2,3,...]

```
nasty : nat -> nat stream
nasty n = if n < 2
          then Cons n (nasty (n+1))
          else inc (tail (nasty n))
```

A PL of course has no problem with this definition

The Higher Responsibility of Proof Assistants

```
inc : nat stream -> nat stream
```

Example: inc [0,1,2,...] = [1,2,3,...]

```
nasty : nat -> nat stream
nasty n = if n < 2
          then Cons n (nasty (n+1))
          else inc (tail (nasty n))
```

A PL of course has no problem with this definition

However, in a (total-logic) PA:

```
nasty 2 = inc (tail (nasty 1)) =
inc (tail (Cons 1 (nasty 2))) = inc (nasty 2)
```

The Higher Responsibility of Proof Assistants

How can a PA accept

```
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

but reject nasty?

The Higher Responsibility of Proof Assistants

How can a PA accept

```
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

but reject nasty?

1. Syntactic check: Reject both fib and nasty (ask the user to rephrase fib) (e.g., Coq)

The Higher Responsibility of Proof Assistants

How can a PA accept

```
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

but reject nasty?

1. Syntactic check: Reject both fib and nasty (ask the user to rephrase fib) (e.g., Coq)
2. Require "size" annotations to convince the system that fib is productive (e.g., Agda)

The Higher Responsibility of Proof Assistants

How can a PA accept

```
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

but reject nasty?

1. Syntactic check: Reject both fib and nasty (ask the user to rephrase fib) (e.g., Coq)
2. Require "size" annotations to convince the system that fib is productive (e.g., Agda)
3. Our approach (in Isabelle/HOL)

The Higher Responsibility of Proof Assistants

How can a PA accept

```
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

but reject nasty?

1. Syntactic check: Reject both fib and nasty (ask the user to rephrase fib) (e.g., Coq)

2. Require "size" annotations to convince the system that fib is productive (e.g., Agda)

3. Our approach (in Isabelle/HOL)

Compile the definition into a low-level non-recursive definition using a smart corecursor

The Higher Responsibility of Proof Assistants

How can a PA accept

```
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

but reject nasty?

1. Syntactic check: Reject both fib and nasty (ask the user to rephrase fib) (e.g., Coq)
2. Require "size" annotations to convince the system that fib is productive (e.g., Agda)
3. Our approach (in Isabelle/HOL)

Compile the definition into a low-level non-recursive definition using a smart corecursor

Train the system with each new definition to make the corecursor smarter

The Higher Responsibility of Proof Assistants

How can a PA accept

```
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

but reject nasty?

1. Syntactic check: Reject both fib and nasty (ask the user to rephrase fib) (e.g., Coq)
2. Require "size" annotations to convince the system that fib is productive (e.g., Agda)
3. Our approach (in Isabelle/HOL)

SAFETY — Compile the definition into a low-level non-recursive definition using a smart corecursor

FLEXIBILITY — Train the system with each new definition to make the corecursor smarter

LCF Philosophy

(1) Why introduce concepts as new primitives when you can reduce them to existing primitives?

LCF Philosophy

(1) Why introduce concepts as new primitives when you can reduce them to existing primitives?

(2) Why invent a new logic CoolL when you can use an existing logic GoodOldL?

LCF Philosophy

(1) Why introduce concepts as new primitives when you can reduce them to existing primitives?

(2) Why invent a new logic CoolL when you can use an existing logic GoodOldL?

Well, GoodOldL is not convenient for proof developments.

LCF Philosophy

(1) Why introduce concepts as new primitives when you can reduce them to existing primitives?

(2) Why invent a new logic CoolL when you can use an existing logic GoodOldL?

Well, GoodOldL is not convenient for proof developments.

OK, but why not reduce the CoolL primitives to GoodOldL?

LCF Philosophy

(1) Why introduce concepts as new primitives when you can reduce them to existing primitives?

(2) Why invent a new logic CoolL when you can use an existing logic GoodOldL?

Well, GoodOldL is not convenient for proof developments.

OK, but why not reduce the CoolL primitives to GoodOldL?

This would be a lot of work.

LCF Philosophy

(1) Why introduce concepts as new primitives when you can <u>reduce</u> them to existing primitives?

(2) Why invent a new logic CoolL when you can use an existing logic GoodOldL?

Well, GoodOldL is not convenient for proof developments.

OK, but why not reduce the CoolL primitives to GoodOldL?

This would be a lot of work.

Yes, but at least GoodOldL is (very probably) consistent.

LCF Philosophy

(1) Why introduce concepts as new primitives when you can <u>reduce</u> them to existing primitives?

(2) Why invent a new logic CoolL when you can use an existing logic GoodOldL?

Well, GoodOldL is not convenient for proof developments.

OK, but why not reduce the CoolL primitives to GoodOldL?

This would be a lot of work.

Yes, but at least GoodOldL is (very probably) consistent.
Fixing CoolL inconsistency problems will be even more work!

Witnessing (Co)Datatypes

Four years of work

Witnessing (Co)Datatypes

Four years of work

LICS'12, ITP'14($\times 2$), IJCAR'14, ESOP'15

Witnessing (Co)Datatypes

Four years of work

LICS'12, ITP'14($\times 2$), IJCAR'14, ESOP'15
Recent unpublished work: ICFP'15 submission

Witnessing (Co)Datatypes

Four years of work

LICS'12, ITP'14($\times 2$), IJCAR'14, ESOP'15
Recent unpublished work: ICFP'15 submission

Flexible mechanism for
(co)inductive and (co)recursive specifications

Witnessing (Co)Datatypes

Four years of work

LICS'12, ITP'14($\times 2$), IJCAR'14, ESOP'15
Recent unpublished work: ICFP'15 submission

Flexible mechanism for
(co)inductive and (co)recursive specifications

Entirely reduced to the primitives of our
GoodOldL $=$ Higher-Order Logic

(Co)Datatypes in Isabelle/HOL

```
codatatype α stream = Cons α (α stream)
```

```
corec + : nat stream -> nat stream -> nat stream
xs + ys = Cons (hd xs + hd ys) (tl xs + tl ys)
```

corec_friedly $+$

```
corec fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

(Co)Datatypes in Isabelle/HOL

codatatype α stream = Cons α (α stream)

α stream = GFP (Λ β. α $\times$ β)

```
corec + : nat stream -> nat stream -> nat stream
xs + ys = Cons (hd xs + hd ys) (tl xs + tl ys)
```

corec_friedly $+$

```
corec fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

(Co)Datatypes in Isabelle/HOL

```
codatatype α stream = Cons α (α stream)
```

```
α stream = GFP (Λ β. α × β)
```

```
streamCorec : (β -> α × β) -> β -> α stream
```

```
corec + : nat stream -> nat stream -> nat stream
xs + ys = Cons (hd xs + hd ys) (tl xs + tl ys)
```

corec_friedly $+$

```
corec fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

(Co)Datatypes in Isabelle/HOL

```
codatatype α stream = Cons α (α stream)
```

```
α stream = GFP (Λ β. α × β)
```

```
streamCorec : (β -> α × β) -> β -> α stream
```

```
corec + : nat stream -> nat stream -> nat stream
xs + ys = Cons (hd xs + hd ys) (tl xs + tl ys)
```

$+ =$ streamCorec (λ (xs,ys). (hd xs $+$ hd ys, (tl xs, tl ys)))

corec_friedly $+$

```
corec fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

(Co)Datatypes in Isabelle/HOL

```
codatatype α stream = Cons α (α stream)
```

```
α stream = GFP (Λ β. α × β)
```

```
streamCorec : (β -> α × β) -> β -> α stream
```

```
corec + : nat stream -> nat stream -> nat stream
xs + ys = Cons (hd xs + hd ys) (tl xs + tl ys)
```

$+ =$ streamCorec (λ (xs,ys). (hd xs $+$ hd ys, (tl xs, tl ys)))

For now, streamCorec is “primitive”.
corec_friedly $+$

```
corec fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

(Co)Datatypes in Isabelle/HOL

```
codatatype α stream = Cons α (α stream)
```

```
α stream = GFP (Λ β. α × β)
```

```
streamCorec : (β -> α × β) -> β -> α stream
```

```
corec + : nat stream -> nat stream -> nat stream
xs + ys = Cons (hd xs + hd ys) (tl xs + tl ys)
```

$+ =$ streamCorec (λ (xs,ys). (hd xs $+$ hd ys, (tl xs, tl ys)))

For now, streamCorec is "primitive". But it evolves!
corec_friedly $+$

```
corec fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

(Co)Datatypes in Isabelle/HOL

```
codatatype α stream = Cons α (α stream)
```

α `stream = GFP (`$\Lambda\ \beta.\ \alpha \times \beta$`)`

`streamCorec : (`β `->` $\alpha \times \beta$`) ->` β `->` α `stream`

```
corec + : nat stream -> nat stream -> nat stream
xs + ys = Cons (hd xs + hd ys) (tl xs + tl ys)
```

$+ =$ streamCorec (λ (xs,ys). (hd xs $+$ hd ys, (tl xs, tl ys)))

For now, streamCorec is "primitive". But it evolves!
corec_friedly $+$ Parametricity proof

```
corec fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

(Co)Datatypes in Isabelle/HOL

```
codatatype α stream = Cons α (α stream)
```

```
α stream = GFP (Λ β. α × β)
```

```
streamCorec : (β -> α×β) -> β -> α stream
```

```
corec + : nat stream -> nat stream -> nat stream
xs + ys = Cons (hd xs + hd ys) (tl xs + tl ys)
```

$+ = \text{streamCorec}\ (\lambda\ (\text{xs,ys}).\ (\text{hd xs} + \text{hd ys},\ (\text{tl xs},\ \text{tl ys})))$

For now, streamCorec is "primitive". But it evolves!
corec_friedly +

streamCorec : (β -> $\mathsf{T}_{\mathsf{Cons},+}(\alpha \times \mathsf{T}_{\mathsf{Cons},+}(\beta))$) -> β -> α stream

```
corec fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

(Co)Datatypes in Isabelle/HOL

`codatatype` α `stream = Cons` α `(`α `stream)`

α `stream = GFP (`Λ β`.` α $\times$ β`)`

`streamCorec : (`β `->` $\alpha \times \beta$`) ->` β `->` α `stream`

```
corec + : nat stream -> nat stream -> nat stream
xs + ys = Cons (hd xs + hd ys) (tl xs + tl ys)
```

$+ =$ streamCorec $(\lambda$ (xs,ys). (hd xs $+$ hd ys, (tl xs, tl ys)))

For now, streamCorec is “primitive”. But it evolves!
corec_friedly $+$

`streamCorec : (`β `->` $\mathsf{T}_{\mathsf{Cons},+}(\alpha \times \mathsf{T}_{\mathsf{Cons},+}(\beta))$`) ->` β `->` α `stream`

```
corec fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

fib $=$ streamCorec $(\lambda u.\ \mathsf{Cons}\ 0\ (1, u) + (0, u))$

(Co)Datatypes in Isabelle/HOL

```
codatatype α stream = Cons α (α stream)
```

```
α stream = GFP (Λ β. α × β)
```

```
streamCorec : (β -> α × β) -> β -> α stream
```

```
corec + : nat stream -> nat stream -> nat stream
xs + ys = Cons (hd xs + hd ys) (tl xs + tl ys)
```

$+ = \text{streamCorec}\ (\lambda\ (\text{xs,ys}).\ (\text{hd xs} + \text{hd ys},\ (\text{tl xs},\ \text{tl ys})))$

For now, streamCorec is "primitive". But it evolves!
corec_friedly $+$

streamCorec : $(\beta$ -> $\mathsf{T}_{\mathsf{Cons},+}(\alpha \times \mathsf{T}_{\mathsf{Cons},+}(\beta)))$ -> β -> α stream

```
corec fib : nat stream
fib = Cons 0 (Cons 1 fib) + Cons 0 fib
```

$\text{fib} = \text{streamCorec}\ (\lambda u.\ \text{Cons}\ 0\ (1, u) + (0, u))$

(Co)Datatypes in Isabelle/HOL

codatatype α stream = Cons α (α stream)

α stream = GFP ($\Lambda\ \beta.\ \alpha \times \beta$)

(Co)Datatypes in Isabelle/HOL

`codatatype` α `stream = Cons` α `(`α `stream)`

α `stream = GFP (`$\Lambda\ \beta.\ \alpha \times \beta$`)`

For Haskell, and indeed for most PAs, $\times$ is “just” a type constructor

(Co)Datatypes in Isabelle/HOL

`codatatype` α `stream = Cons` α `(`α `stream)`

α `stream = GFP (`$\Lambda\ \beta.\ \alpha \times \beta$`)`

For Haskell, and indeed for most PAs, $\times$ is “just” a type constructor

For Isabelle/HOL, $\times$ is much more:

(Co)Datatypes in Isabelle/HOL

`codatatype` α `stream = Cons` α `(`α `stream)`

α `stream = GFP (`$\Lambda\ \beta.\ \alpha \times \beta$`)`

For Haskell, and indeed for most PAs, $\times$ is "just" a type constructor

For Isabelle/HOL, $\times$ is much more:

- A mapper $\mathsf{map}_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$

(Co)Datatypes in Isabelle/HOL

`codatatype` α `stream = Cons` α `(`α `stream)`

α `stream = GFP (`Λ β`.` α $\times$ β`)`

For Haskell, and indeed for most PAs, $\times$ is "just" a type constructor

For Isabelle/HOL, $\times$ is much more:

- A mapper $\mathsf{map}_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$
- A relator $\mathsf{rel}_\times : (\alpha_1 \to \alpha_2 \to \mathsf{bool}) \to (\beta_1 \to \beta_2 \to \mathsf{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \mathsf{bool}$

(Co)Datatypes in Isabelle/HOL

```
codatatype α stream = Cons α (α stream)
```

```
α stream = GFP (Λ β. α × β)
```

For Haskell, and indeed for most PAs, $\times$ is "just" a type constructor

For Isabelle/HOL, $\times$ is much more:

- A mapper $\mathsf{map}_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$
- A relator $\mathsf{rel}_\times : (\alpha_1 \to \alpha_2 \to \mathsf{bool}) \to (\beta_1 \to \beta_2 \to \mathsf{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \mathsf{bool}$
- Nonemptiness witnesses $\mathsf{wit}_\times : \alpha \to \beta \to \alpha \times \beta$

(Co)Datatypes in Isabelle/HOL

```
codatatype α stream = Cons α (α stream)
```

```
α stream = GFP (Λ β. α × β)
```

For Haskell, and indeed for most PAs, $\times$ is "just" a type constructor

For Isabelle/HOL, $\times$ is much more:

- A mapper $\mathsf{map}_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$
- A relator $\mathsf{rel}_\times : (\alpha_1 \to \alpha_2 \to \mathsf{bool}) \to (\beta_1 \to \beta_2 \to \mathsf{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \mathsf{bool}$
- Nonemptiness witnesses $\mathsf{wit}_\times : \alpha \to \beta \to \alpha \times \beta$
- A cardinal bound

(Co)Datatypes in Isabelle/HOL

`codatatype` α `stream = Cons` α `(`α `stream)`

α `stream = GFP (`$\Lambda\ \beta.\ \alpha \times \beta$`)`

For Haskell, and indeed for most PAs, $\times$ is "just" a type constructor

For Isabelle/HOL, $\times$ is much more:

- A mapper $\mathsf{map}_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$
- A relator $\mathsf{rel}_\times : (\alpha_1 \to \alpha_2 \to \mathsf{bool}) \to (\beta_1 \to \beta_2 \to \mathsf{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \mathsf{bool}$
- Nonemptiness witnesses $\mathsf{wit}_\times : \alpha \to \beta \to \alpha \times \beta$
- A cardinal bound

Bounded Natural Functor (BNF)

(Co)Datatypes in Isabelle/HOL

`codatatype` α `stream = Cons` α `(`α `stream)`

α `stream = GFP (`$\Lambda\ \beta.\ \alpha \times \beta$`)`

For Haskell, and indeed for most PAs, $\times$ is "just" a type constructor

For Isabelle/HOL, $\times$ is much more:

- A mapper $\mathsf{map}_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$
- A relator $\mathsf{rel}_\times : (\alpha_1 \to \alpha_2 \to \mathsf{bool}) \to (\beta_1 \to \beta_2 \to \mathsf{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \mathsf{bool}$
- Nonemptiness witnesses $\mathsf{wit}_\times : \alpha \to \beta \to \alpha \times \beta$
- A cardinal bound

Bounded Natural Functor (BNF)

(Co)Datatypes in Isabelle/HOL

```
codatatype α stream = Cons α (α stream)
```

```
α stream = GFP (Λ β. α × β)
```

For Haskell, and indeed for most PAs, $\times$ is "just" a type constructor

For Isabelle/HOL, $\times$ is much more:

- A mapper $\mathsf{map}_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$
- A relator $\mathsf{rel}_\times : (\alpha_1 \to \alpha_2 \to \mathsf{bool}) \to (\beta_1 \to \beta_2 \to \mathsf{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \mathsf{bool}$
- Nonemptiness witnesses $\mathsf{wit}_\times : \alpha \to \beta \to \alpha \times \beta$
- A cardinal bound

Bounded Natural Functor (BNF)

```
datatype α tree = Leaf α | Node (α tree stream)
```

(Co)Datatypes in Isabelle/HOL

`codatatype` α `stream = Cons` α `(`α `stream)`

α `stream = GFP (`$\Lambda\ \beta.\ \alpha \times \beta$`)`

For Haskell, and indeed for most PAs, $\times$ is "just" a type constructor

For Isabelle/HOL, $\times$ is much more:

- A mapper $\mathsf{map}_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$
- A relator $\mathsf{rel}_\times : (\alpha_1 \to \alpha_2 \to \mathsf{bool}) \to (\beta_1 \to \beta_2 \to \mathsf{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \mathsf{bool}$
- Nonemptiness witnesses $\mathsf{wit}_\times : \alpha \to \beta \to \alpha \times \beta$
- A cardinal bound

Bounded Natural Functor (BNF)

`datatype` α `tree = Leaf` α $|$ `Node (`α `tree stream)`

α `tree = LFP (`$\Lambda\ \beta.\ \alpha + \beta\ \mathsf{stream}$`)`

(Co)Datatypes in Isabelle/HOL

`codatatype` α `stream = Cons` α `(`α `stream)`

α `stream = GFP (`$\Lambda\ \beta.\ \alpha \times \beta$`)`

For Haskell, and indeed for most PAs, $\times$ is "just" a type constructor

For Isabelle/HOL, $\times$ is much more:

- A mapper $\mathsf{map}_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$
- A relator $\mathsf{rel}_\times : (\alpha_1 \to \alpha_2 \to \mathsf{bool}) \to (\beta_1 \to \beta_2 \to \mathsf{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \mathsf{bool}$
- Nonemptiness witnesses $\mathsf{wit}_\times : \alpha \to \beta \to \alpha \times \beta$
- A cardinal bound

Bounded Natural Functor (BNF)

`datatype` α `tree = Leaf` α $|$ `Node (`α `tree list)`

α `tree = LFP (`$\Lambda\ \beta.\ \alpha + \beta$ list`)`

(Co)Datatypes in Isabelle/HOL

`codatatype` α `stream = Cons` α `(`α `stream)`

α `stream = GFP (`$\Lambda\ \beta.\ \alpha \times \beta$`)`

For Haskell, and indeed for most PAs, $\times$ is "just" a type constructor

For Isabelle/HOL, $\times$ is much more:

- A mapper $\mathsf{map}_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$
- A relator $\mathsf{rel}_\times : (\alpha_1 \to \alpha_2 \to \mathsf{bool}) \to (\beta_1 \to \beta_2 \to \mathsf{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \mathsf{bool}$
- Nonemptiness witnesses $\mathsf{wit}_\times : \alpha \to \beta \to \alpha \times \beta$
- A cardinal bound

Bounded Natural Functor (BNF)

`datatype` α `tree = Leaf` α `| Node (`α `tree countable_set)`

α `tree = LFP (`$\Lambda\ \beta.\ \alpha + \beta$ `countable_set)`

(Co)Datatypes in Isabelle/HOL

`codatatype` α `stream = Cons` α `(`α `stream)`

α `stream = GFP (`$\Lambda\ \beta.\ \alpha \times \beta$`)`

For Haskell, and indeed for most PAs, $\times$ is "just" a type constructor

For Isabelle/HOL, $\times$ is much more:

- A mapper $\mathsf{map}_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$
- A relator $\mathsf{rel}_\times : (\alpha_1 \to \alpha_2 \to \mathsf{bool}) \to (\beta_1 \to \beta_2 \to \mathsf{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \mathsf{bool}$
- Nonemptiness witnesses $\mathsf{wit}_\times : \alpha \to \beta \to \alpha \times \beta$
- A cardinal bound

Bounded Natural Functor (BNF)

`datatype` α `tree = Leaf` α | `Node (`α `tree bag)`

α `tree = LFP (`$\Lambda\ \beta.\ \alpha + \beta$ `bag)`

(Co)Datatypes in Isabelle/HOL

`codatatype` α `stream = Cons` α `(`α `stream)`

α `stream = GFP (`$\Lambda\ \beta.\ \alpha \times \beta$`)`

For Haskell, and indeed for most PAs, $\times$ is "just" a type constructor

For Isabelle/HOL, $\times$ is much more:

- A mapper $\mathsf{map}_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$
- A relator $\mathsf{rel}_\times : (\alpha_1 \to \alpha_2 \to \mathsf{bool}) \to (\beta_1 \to \beta_2 \to \mathsf{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \mathsf{bool}$
- Nonemptiness witnesses $\mathsf{wit}_\times : \alpha \to \beta \to \alpha \times \beta$
- A cardinal bound

Bounded Natural Functor (BNF)

`datatype` α `tree = Leaf` α $|$ `Node (`α `tree PLUG_YOUR_OWN)`

α `tree = LFP (`$\Lambda\ \beta.\ \alpha + \beta$ `PLUG_YOUR_OWN)`

Witnessing (Co)Datatypes

Isabelle maintains Bounded Natural Functors

Witnessing (Co)Datatypes

Isabelle maintains Bounded Natural Functors

$$\Downarrow$$

Modular, Open-Ended (Co)Datatypes

Witnessing (Co)Datatypes

Isabelle maintains Bounded Natural Functors

$$\Downarrow$$

Modular, Open-Ended (Co)Datatypes

$$\Downarrow$$

Safe and Flexible (Co)Recursive Definitions

Related Work

Inspiring Work

- Paulson's pioneering fixpoint constructions in Isabelle/ZF

Related Work

Inspiring Work

- Paulson's pioneering fixpoint constructions in Isabelle/ZF
- Containers (Abbott, Altenkirch, Ghani)
- Fibrations (Hermida, Jacobs)
- Distributive Laws (Turi and Plotkin, Bartels, Jacobs, Milius, Hinze, etc.)
- Coinduction Up-To (Rot, Bonsangue, Rutten, Silva, etc.)

Related Work

Inspiring Work

- Paulson's pioneering fixpoint constructions in Isabelle/ZF
- Containers (Abbott, Altenkirch, Ghani)
- Fibrations (Hermida, Jacobs)
- Distributive Laws (Turi and Plotkin, Bartels, Jacobs, Milius, Hinze, etc.)
- Coinduction Up-To (Rot, Bonsangue, Rutten, Silva, etc.)

Competing (and Inspiring) Work

- Sized types in MiniAgda, Agda (Abel)
- Clock Variables (Atkey and McBride, Clouston et al.)

Witnessing (Co)Datatypes

Please try our new (co)datatypes in Isabelle/HOL:
you won't regret it. ☺