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)

Proof Assistant (PA)

Automatic Code Generation
Proof Assistants versus Programming Languages

Programming Language (PL)

Proof Assistant (PA)

Automatic Code Generation

PA = Smart PL
Proof Assistants are Smart

\[ \text{fact} : \text{nat} \rightarrow \text{nat} \]
\[ \text{fact} \ n = \text{case} \ n \ of \ 0 \Rightarrow 1 \]
\[ \quad \mid \ \text{Suc} \ m \Rightarrow n \times \text{fact} \ m \]
Proof Assistants are Smart

\[
\text{fact} : \text{naut} \rightarrow \text{nat} \\
\text{fact } n = \text{case } n \text{ of } 0 \Rightarrow 1 \\
\quad \mid \text{Suc } m \Rightarrow n \ast \text{fact } m
\]

For a PL, this is just a partial function
Proof Assistants are Smart

```haskell
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

\[
\text{fact : nat } \rightarrow \text{ nat} \\
\text{fact } n = \text{ case } n \text{ of } 0 \Rightarrow 1 \\
\hspace{1cm} \mid \text{ Suc } m \Rightarrow n \times \text{ fact } m
\]

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

\[
\text{+ : nat stream } \rightarrow \text{ nat stream} \\
\text{fib : nat stream} \\
\text{fib } = \text{ Cons } 0 \ (\text{Cons } 1 \ \text{fib}) \ + \ \text{Cons } 0 \ \text{fib}
\]
Proof Assistants are Smart

\[
\text{fact} : \text{nat} \rightarrow \text{nat} \\
\text{fact } n = \text{case } n \text{ of } 0 \Rightarrow 1 \\
\phantom{\text{fact } n = \text{case } n \text{ of } 0 \Rightarrow 1} | \text{Suc } m \Rightarrow n \times \text{fact } m
\]

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

\[
+ : \text{nat stream} \rightarrow \text{nat stream} \rightarrow \text{nat stream}
\]

\[
\text{fib} : \text{nat stream} \\
\text{fib} = \text{Cons } 0 (\text{Cons } 1 \text{ fib}) + \text{Cons } 0 \text{ fib}
\]

Haskell of course accepts this.
Proof Assistants are Smart

\[ \text{fact} : \text{nat} \to \text{nat} \]
\[ \text{fact} \ n = \text{case} \ n \text{ of} \ 0 \Rightarrow 1 \]
\[ \quad \mid \text{Suc} \ m \Rightarrow n \ast \text{fact} \ m \]

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

\[ + : \text{nat stream} \to \text{nat stream} \to \text{nat stream} \]

\[ \text{fib} : \text{nat stream} \]
\[ \text{fib} = \text{Cons} \ 0 \ (\text{Cons} \ 1 \ \text{fib}) + \text{Cons} \ 0 \ \text{fib} \]

Haskell of course accepts this. But is it productive?
Fact Assistants are Smart

\[
\text{fact} : \text{nat} \rightarrow \text{nat} \\
\text{fact } n = \text{case } n \text{ of } 0 \Rightarrow 1 \\
\quad \quad \quad \quad \mid \text{Suc } m \Rightarrow n \ast \text{fact } m
\]

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

\[
+ : \text{nat stream} \rightarrow \text{nat stream} \rightarrow \text{nat stream}
\]

\[
\text{fib} : \text{nat stream}
\]

\[
\text{fib} = \text{Cons } 0 \ (\text{Cons } 1 \ \text{fib}) + \text{Cons } 0 \ \text{fib}
\]

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

\[ \text{inc} : \text{nat stream} \rightarrow \text{nat stream} \]

Example: \( \text{inc} \{0,1,2,\ldots\} = \{1,2,3,\ldots\} \)

\[ \text{nasty} : \text{nat} \rightarrow \text{nat stream} \]
\[ \text{nasty} \ n = \begin{cases} 
\text{Cons} \ n \ (\text{nasty} \ (n+1)) & \text{if } n < 2 \\
\text{inc} \ (\text{tail} \ (\text{nasty} \ n)) & \text{else}
\end{cases} \]

A PL of course has no problem with this definition.

However, in a (total-logic) PA:

\[ \text{nasty} \ 2 = \text{inc} \ (\text{tail} \ (\text{nasty} \ 1)) = \text{inc} \ (\text{tail} \ (\text{Cons} \ 1 \ (\text{nasty} \ 2))) = \text{inc} \ (\text{nasty} \ 2) \]
The Higher Responsibility of Proof Assistants

\text{inc} : \text{nat stream} \rightarrow \text{nat stream}

Example: \text{inc} [0,1,2,...] = [1,2,3,...]

nasty : \text{nat} \rightarrow \text{nat stream}

\text{nasty} \; n = \begin{cases} 
\text{Cons} \; n \; \text{\text{nasty} \; (n+1)} & \text{if} \; n < 2 \\
\text{inc} \; (\text{tail} \; \text{nasty} \; n) & \text{else}
\end{cases}

A PL of course has no problem with this definition
However, in a (total-logic) PA:
\text{nasty} \; 2 = \text{inc} \; (\text{tail} \; \text{nasty} \; 1) = \text{inc} \; (\text{tail} \; \text{Cons} \; 1 \; \text{nasty} \; 2) = \text{inc} \; \text{nasty} \; 2
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

\( \text{inc} : \text{nat stream} \rightarrow \text{nat stream} \)

Example: \( \text{inc } [0,1,2,...] = [1,2,3,...] \)

\( \text{nasty} : \text{nat} \rightarrow \text{nat stream} \)
\( \text{nasty } n = \text{if } n < 2 \)
\( \quad \text{then Cons } n \ (\text{nasty } (n+1)) \)
\( \quad \text{else inc } (\text{tail } (\text{nasty } n)) \)

A PL of course has no problem with this definition
The Higher Responsibility of Proof Assistants

\[
\text{inc} : \text{nat stream} \rightarrow \text{nat stream}
\]

Example: \(\text{inc} [0,1,2,...] = [1,2,3,...]\)

\[
\text{nasty} : \text{nat} \rightarrow \text{nat stream}
\]

\[
\text{nasty} \ n = \text{if } n < 2 \text{ then } \text{Cons} \ n \ (\text{nasty} \ (n+1)) \text{ else inc (tail (nasty n))}
\]

A PL of course has no problem with this definition

However, in a (total-logic) PA:

\[
\text{nasty} \ 2 = \text{inc} \ (\text{tail} \ (\text{nasty} \ 1)) = \\
\text{inc} \ (\text{tail} \ (\text{Cons} \ 1 \ (\text{nasty} \ 2))) = \text{inc} \ (\text{nasty} \ 2)
\]
The Higher Responsibility of Proof Assistants

How can a PA accept

\[ \text{fib} = \text{Cons 0 (Cons 1 fib)} + \text{Cons 0 fib} \]

but reject \text{nasty}?

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

2. Require "size" annotations to convince the system that \text{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
The Higher Responsibility of Proof Assistants

How can a PA accept

\[ \text{fib} = \text{Cons } 0 (\text{Cons } 1 \text{ fib}) + \text{Cons } 0 \text{ fib} \]

but reject \text{nasty}?

1. Syntactic check: Reject both \text{fib} and \text{nasty} (ask the user to rephrase \text{fib}) (e.g., Coq)
The Higher Responsibility of Proof Assistants

How can a PA accept

\[ \text{fib} = \text{Cons} \ 0 \ (\text{Cons} \ 1 \ \text{fib}) + \text{Cons} \ 0 \ \text{fib} \]

but reject \text{nasty}?

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

2. Require “size” annotations to convince the system that
\text{fib} is productive (e.g., Agda)
The Higher Responsibility of Proof Assistants

How can a PA accept

\[ \text{fib} = \text{Cons 0 } (\text{Cons 1 fib}) + \text{Cons 0 fib} \]

but reject \text{nasty}? 

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

2. Require “size” annotations to convince the system that \text{fib} is productive (e.g., Agda)

3. Our approach (in Isabelle/HOL)
The Higher Responsibility of Proof Assistants

How can a PA accept

\( \text{fib} = \text{Cons 0 (Cons 1 fib)} + \text{Cons 0 fib} \)

but reject \( \text{nasty} \)?

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

2. Require “size” annotations to convince the system that \( \text{fib} \) 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

\[ \text{fib} = \text{Cons 0 (Cons 1 fib)} + \text{Cons 0 fib} \]

but reject \text{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 \textit{smart corecursor} |
| FLEXIBILITY    | \textit{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?
(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?
(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.
(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.

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

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(×2), IJCAR’14, ESOP’15
Witnessing (Co)Datatypes

Four years of work

LICS’12, ITP’14(×2), IJCAR’14, ESOP’15
Recent unpublished work: ICFP’15 submission
Witnessing (Co)Datatypes

Four years of work

LICS’12, ITP’14(×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(×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 × 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 \( \alpha \) stream = Cons \( \alpha \) (\( \alpha \) stream)

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

streamCorec : (\( \beta \) -> \( \alpha \times \beta \)) -> \( \beta \) -> \( \alpha \) stream

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

corec_friendly +

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 $\alpha$ stream = Cons $\alpha$ ($\alpha$ stream)

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

streamCorec : ($\beta$ -> $\alpha \times \beta$) -> $\beta$ -> $\alpha$ 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”.
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_friendly +

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


\textbf{(Co)Datatypes in Isabelle/HOL}

\begin{itemize}
  \item \texttt{codatatype } \( \alpha \) stream = Cons \( \alpha \) (\( \alpha \) stream)
  \item \( \alpha \) stream = GFP (\( \Lambda \beta \). \( \alpha \times \beta \))
  \item \texttt{streamCorec} : (\( \beta \rightarrow \alpha \times \beta \)) -> \( \beta \rightarrow \alpha \) stream
  \item \texttt{corec +} : nat stream -> nat stream -> nat stream
  \item \( \texttt{xs + ys} = \text{Cons} \((\text{hd} \text{ xs + hd} \text{ ys})\) (\text{tl} \text{ xs + tl} \text{ ys})\)
  \item \( + = \texttt{streamCorec} (\lambda (xs,ys). (\text{hd} \text{ xs + hd} \text{ ys, (tl} \text{ xs, tl} \text{ ys)}))\)
\end{itemize}

For now, \texttt{streamCorec} is “primitive”. But it evolves!
\begin{itemize}
  \item \texttt{corec_friedly +} \quad \text{Parametricity proof}
\end{itemize}

\begin{itemize}
  \item \texttt{corec fib} : nat stream
  \item \( \texttt{fib} = \text{Cons} 0 \ (\text{Cons} 1 \ \texttt{fib}) + \text{Cons} 0 \ \texttt{fib} \)
\end{itemize}
(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_friendly +

streamCorec : (β -> T_{Cons,+}(α × T_{Cons,+}(β))) -> 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 × 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 +

streamCorec : (β -> T_{Cons,+}(α × T_{Cons,+}(β))) -> β -> α stream

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

fib = streamCorec (λ u. Cons 0 (1, u) + (0, u))
(Co)Datatypes in Isabelle/HOL

codatatype \( \alpha \) stream = Cons \( \alpha \) (\( \alpha \) stream)

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

streamCorec : (\( \beta \) \rightarrow \( \alpha \) \times \( \beta \)) \rightarrow \( \beta \) \rightarrow \( \alpha \) stream

corec + : nat stream \rightarrow nat stream \rightarrow 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_friendly +

streamCorec : (\( \beta \) \rightarrow T_{\text{Cons},+}(\( \alpha \) \times T_{\text{Cons},+}(\( \beta \)))) \rightarrow \( \beta \) \rightarrow \( \alpha \) stream

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

fib = streamCorec (\( \lambda u \). 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)
(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, × is “just” a type constructor
(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, × is “just” a type constructor.

For Isabelle/HOL, × is much more:
(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, × is “just” a type constructor

For Isabelle/HOL, × is much more:

• A mapper \( \text{map}_\times: (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \)
 Codatatypes in Isabelle/HOL

codatatype $\alpha$ stream = Cons $\alpha$ ($\alpha$ stream)

$\alpha$ 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 $\text{map}_\times : (\alpha_1 \rightarrow \alpha_2) \rightarrow (\beta_1 \rightarrow \beta_2) \rightarrow \alpha_1 \times \beta_1 \rightarrow \alpha_2 \times \beta_2$
- A relator $\text{rel}_\times : (\alpha_1 \rightarrow \alpha_2 \rightarrow \text{bool}) \rightarrow (\beta_1 \rightarrow \beta_2 \rightarrow \text{bool}) \rightarrow \alpha_1 \times \beta_1 \rightarrow \alpha_2 \times \beta_2 \rightarrow \text{bool}$
(Co)Datatypes in Isabelle/HOL

codatatype $\alpha$ stream = Cons $\alpha$ ($\alpha$ stream)

$\alpha$ 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 $\text{map}_x : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$
- A relator $\text{rel}_x : (\alpha_1 \to \alpha_2 \to \text{bool}) \to (\beta_1 \to \beta_2 \to \text{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \text{bool}$
- Nonemptiness witnesses $\text{wit}_x : \alpha \to \beta \to \alpha \times \beta$
(Co)Datatypes in Isabelle/HOL

codatatype $\alpha$ stream = Cons $\alpha$ ($\alpha$ stream)

$\alpha$ 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 $\text{map}_x : (\alpha_1 \rightarrow \alpha_2) \rightarrow (\beta_1 \rightarrow \beta_2) \rightarrow \alpha_1 \times \beta_1 \rightarrow \alpha_2 \times \beta_2$

- A relator $\text{rel}_x : (\alpha_1 \rightarrow \alpha_2 \rightarrow \text{bool}) \rightarrow (\beta_1 \rightarrow \beta_2 \rightarrow \text{bool}) \rightarrow \alpha_1 \times \beta_1 \rightarrow \alpha_2 \times \beta_2 \rightarrow \text{bool}$

- Nonemptiness witnesses $\text{wit}_x : \alpha \rightarrow \beta \rightarrow \alpha \times \beta$

- A cardinal bound
(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, × is “just” a type constructor

For Isabelle/HOL, × is much more:

- A mapper map_x : (α_1 → α_2) → (β_1 → β_2) → α_1 × β_1 → α_2 × β_2
- A relator rel_x : (α_1 → α_2 → bool) → (β_1 → β_2 → bool) → α_1 × β_1 → α_2 × β_2 → bool
- Nonemptiness witnesses wit_x : α → β → α × β
- 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, × is “just” a type constructor.

For Isabelle/HOL, × is much more:

- A mapper mapₓ : (α₁ → α₂) → (β₁ → β₂) → α₁ × β₁ → α₂ × β₂
- A relator relₓ : (α₁ → α₂ → bool) → (β₁ → β₂ → bool) → α₁ × β₁ → α₂ × β₂ → bool
- Nonemptiness witnesses witₓ : α → β → α × β
- A cardinal bound

Bounded Natural Functor (BNF)
(Co)Datatypes in Isabelle/HOL

codatatype $\alpha$ stream = Cons $\alpha$ ($\alpha$ stream)

$\alpha$ 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 map$_x$ : ($\alpha_1 \rightarrow \alpha_2$) $\rightarrow$ ($\beta_1 \rightarrow \beta_2$) $\rightarrow$ $\alpha_1 \times \beta_1$ $\rightarrow$ $\alpha_2 \times \beta_2$
- A relator rel$_x$ : ($\alpha_1 \rightarrow \alpha_2 \rightarrow$ bool) $\rightarrow$ ($\beta_1 \rightarrow \beta_2 \rightarrow$ bool) $\rightarrow$ $\alpha_1 \times \beta_1$ $\rightarrow$ $\alpha_2 \times \beta_2$ $\rightarrow$ bool
- Nonemptiness witnesses wit$_x$ : $\alpha$ $\rightarrow$ $\beta$ $\rightarrow$ $\alpha \times \beta$
- A cardinal bound

Bounded Natural Functor (BNF)

datatype $\alpha$ tree = Leaf $\alpha$ | Node ($\alpha$ tree stream)
(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, × is “just” a type constructor

For Isabelle/HOL, × is much more:

• A mapper $map_\times : (\alpha_1 \to \alpha_2) \to (\beta_1 \to \beta_2) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2$

• A relator $rel_\times : (\alpha_1 \to \alpha_2 \to \text{bool}) \to (\beta_1 \to \beta_2 \to \text{bool}) \to \alpha_1 \times \beta_1 \to \alpha_2 \times \beta_2 \to \text{bool}$

• Nonemptiness witnesses $wit_\times : \alpha \to \beta \to \alpha \times \beta$

• A cardinal bound

Bounded Natural Functor (BNF)

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

α tree = LFP (Λ β. α + β stream)
(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, × is “just” a type constructor

For Isabelle/HOL, × is much more:

- A mapper map_× : (α₁ → α₂) → (β₁ → β₂) → α₁ × β₁ → α₂ × β₂
- A relator rel_× : (α₁ → α₂ → bool) → (β₁ → β₂ → bool) → α₁ × β₁ → α₂ × β₂ → bool
- Nonemptiness witnesses wit_× : α → β → α × β
- A cardinal bound

Bounded Natural Functor (BNF)

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

α tree = LFP (Λ β. α + β list)
(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, × is “just” a type constructor

For Isabelle/HOL, × is much more:
  - A mapper map_× : (α_1 → α_2) → (β_1 → β_2) → α_1 × β_1 → α_2 × β_2
  - A relator rel_× : (α_1 → α_2 → bool) → (β_1 → β_2 → bool) → α_1 × β_1 → α_2 × β_2 → bool
  - Nonemptiness witnesses wit_× : α → β → α × β
  - A cardinal bound

Bounded Natural Functor (BNF)

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

α tree = LFP (Λ β. α + β countable_set)
(Co)Datatypes in Isabelle/HOL

codatatype $\alpha$ stream = Cons $\alpha$ ($\alpha$ stream)

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

Bounded Natural Functor (BNF)

datatype $\alpha$ tree = Leaf $\alpha$ | Node ($\alpha$ tree bag)

$\alpha$ tree = LFP ($\lambda\beta. \alpha + \beta$ bag)
(Co)Datatypes in Isabelle/HOL

codatatype $\alpha$ stream = Cons $\alpha$ ($\alpha$ stream)

$\alpha$ 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 $\text{map}_x : (\alpha_1 \rightarrow \alpha_2) \rightarrow (\beta_1 \rightarrow \beta_2) \rightarrow \alpha_1 \times \beta_1 \rightarrow \alpha_2 \times \beta_2$
- A relator $\text{rel}_x : (\alpha_1 \rightarrow \alpha_2 \rightarrow \text{bool}) \rightarrow (\beta_1 \rightarrow \beta_2 \rightarrow \text{bool}) \rightarrow \alpha_1 \times \beta_1 \rightarrow \alpha_2 \times \beta_2 \rightarrow \text{bool}$
- Nonemptiness witnesses $\text{wit}_x : \alpha \rightarrow \beta \rightarrow \alpha \times \beta$
- A cardinal bound

Bounded Natural Functor (BNF)

datatype $\alpha$ tree = Leaf $\alpha$ | Node ($\alpha$ tree PLUG_YOUR_OWN)

$\alpha$ 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

⇓

Modular, Open-Ended (Co)Datatypes

⇓

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. 😊