Programming and Reasoning with Infinite Data in Isabelle/HOL

Mathias Fleury

max planck institut informatik

Andreas Lochbihler

ETH zürich

Andrei Popescu

Middlesex University London

You can run our material on your computer

Isabelle: `isabelle.in.tum.de` Theories: `is.gd/Isabelle`

Based on joint work with

Jasmin Blanchette

VU VRIJE UNIVERSITEIT AMSTERDAM

Dmitriy Traytel

ETH zürich

... who have also contributed slides for this presentation

Context

Proof assistants = Tools for

- Defining mathematical concepts
- Proving facts about them

Context

What makes a good proof assistant?

Proof assistants = Tools for

- Defining mathematical concepts
- Proving facts about them

Context

What makes a good proof assistant?

Proof assistants = Tools for

- Defining mathematical concepts
- Proving facts about them Powerful automation

Context

What makes a good proof assistant?

Proof assistants = Tools for

- Defining mathematical concepts — Expressive definitions
- Proving facts about them — Powerful automation

Context

What makes a good proof assistant?

Proof assistants = Tools for

- Defining mathematical concepts — Expressive definitions
- Proving facts about them — Powerful automation

Context

What makes a good proof assistant?

Proof assistants = Tools for

- Defining mathematical concepts — Expressive definitions
- Proving facts about them — Powerful automation

$$\text{collatz} : \text{Nat} \to \text{LazyList}\,(\text{Nat})$$

Context

What makes a good proof assistant?

Proof assistants = Tools for

- Defining mathematical concepts — Expressive definitions
- Proving facts about them — Powerful automation

$$\text{collatz} : \text{Nat} \to \text{LazyList}(\text{Nat})$$

$$\text{collatz}(n) = \begin{cases} [] & \text{if } n \leq 1 \\ \text{collatz}(n \,/\, 2) & \text{if } n > 1 \text{ and } n \text{ even} \\ n \mathbin{\#\#} \text{collatz}(3 * n + 1) & \text{if } n > 1 \text{ and } n \text{ odd} \end{cases}$$

Context

What makes a good proof assistant?

Proof assistants = Tools for

- Defining mathematical concepts — Expressive definitions
- Proving facts about them — Powerful automation

$$\text{collatz} : \boxed{\text{Nat}} \to \text{LazyList}(\text{Nat})$$

$$\text{collatz}(n) = \begin{cases} [] & \text{if } n \leq 1 \\ \text{collatz}(n \,/\, 2) & \text{if } n > 1 \text{ and } n \text{ even} \\ n \mathbin{\#\#} \text{collatz}(3 * n + 1) & \text{if } n > 1 \text{ and } n \text{ odd} \end{cases}$$

recursion

Context

What makes a good proof assistant?

Proof assistants = Tools for

- Defining mathematical concepts — Expressive definitions
- Proving facts about them — Powerful automation

$$\text{collatz} : \boxed{\text{Nat}} \to \boxed{\text{LazyList}\,(\text{Nat})}$$

$$\text{collatz}\,(n) = \begin{cases} [] & \text{if } n \leq 1 \\ \text{collatz}\,(n \,/\, 2) & \text{if } n > 1 \text{ and } n \text{ even} \quad \boxed{\text{recursion}} \\ n \;\#\#\; \text{collatz}\,(3 * n + 1) & \text{if } n > 1 \text{ and } n \text{ odd} \quad \boxed{\text{corecursion}} \end{cases}$$

Big proofs about programs in Isabelle

(an incomplete list)

seL4

Microkernel

Klein et al.

CAVA

LTL model checker

Lammich, Nipkow et al.

Markov_Models

pCTL model checker

Hölzl, Nipkow

Flyspeck

Programs in Hales's proof of the Kepler conjecture

Bauer, Nipkow, Obua

CoCon

Conference management system

Kanav, Lammich, Popescu

PDF-Compiler

Probability density functions compiler

Eberl, Hölzl, Nipkow

JinjaThreads

Java compiler & JMM

Lochbihler

IsaFoR/CeTA

Termination proof certifier

Sternagel, Thiemann et al.

IsaSAT

SAT solver with WL

Fleury, Blanchette et al.

Codatatypes and Corecursion

Codatatypes/Corecursion in Isabelle

Programming Language Examples

Types with infinite values

type	**finite values**	**infinite values**
nat	0, 1, 2, 3, …	
enat	0, 1, 2, 3, …	∞

Types with infinite values

type	**finite values**	**infinite values**
	$0, S(0), S(S(0)), S(S(S(0))), \ldots$	
nat	0, 1, 2, 3, …	
enat	0, 1, 2, 3, …	∞
	$0, S(0), S(S(0)), S(S(S(0))), \ldots$	$S(S(S(S(S(\ldots)))))$

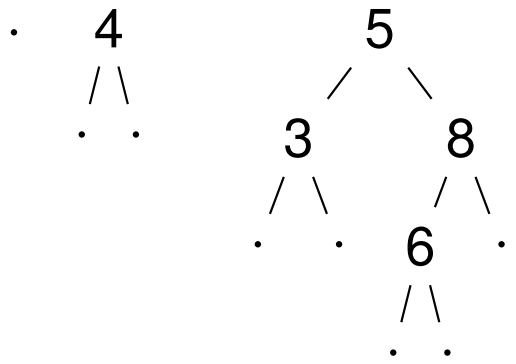
Types with infinite values

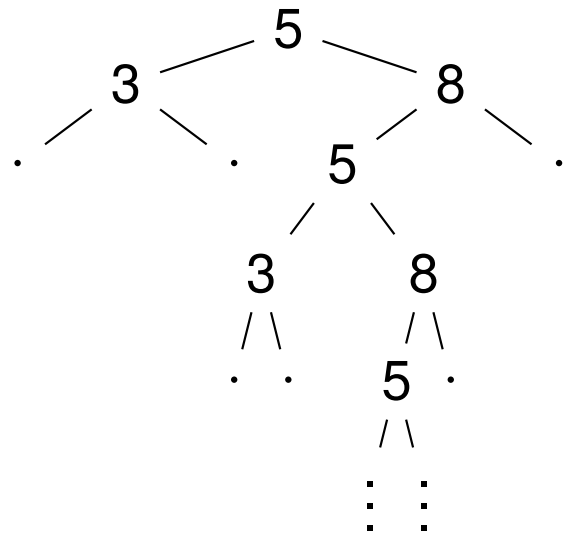
type	**finite values**	**infinite values**
	$0, S(0), S(S(0)), S(S(S(0))), \ldots$	
nat	0, 1, 2, 3, ...	
enat	0, 1, 2, 3, ...	∞
	$0, S(0), S(S(0)), S(S(S(0))), \ldots$	$S(S(S(S(S(\ldots)))))$
list	[], [0], [0,0], [0,1,2,3,4], ...	
stream		[0,0,0,...], [1,2,3,...], [0,1,0,1,...], ...
llist	[], [0], [0,0], [0,1,2,3,4], ...	[0,0,0,...], [1,2,3,...], [0,1,0,1,...], ...

Types with infinite values

type	finite values	infinite values
	$0, S(0), S(S(0)), S(S(S(0))), \ldots$	
nat	0, 1, 2, 3, …	
enat	0, 1, 2, 3, …	∞
	$0, S(0), S(S(0)), S(S(S(0))), \ldots$	$S(S(S(S(S(\ldots)))))$
list	[], [0], [0,0], [0,1,2,3,4], …	
stream		[0,0,0,…], [1,2,3,…], [0,1,0,1,…], …
llist	[], [0], [0,0], [0,1,2,3,4], …	[0,0,0,…], [1,2,3,…], [0,1,0,1,…], …

tree





No recursion on codatatypes

```
datatype   nat  = 0    | Suc  nat
codatatype enat = Zero | eSuc enat
```

Suppose we could do recursion on codatatypes ...

```
  to_nat Zero     = 0
| to_nat (eSuc n) = Suc (to_nat n)
```

No recursion on codatatypes

```
datatype   nat  = 0    | Suc  nat
codatatype enat = Zero | eSuc enat
```

Suppose we could do recursion on codatatypes . . .

```
  to_nat Zero     = 0
| to_nat (eSuc n) = Suc (to_nat n)
```

. . . but codatatypes are not well-founded: ∞ = eSuc ∞

```
to_nat ∞ = to_nat (eSuc ∞) = Suc (to_nat ∞)
```

$$n = 1 + n$$

False

Building infinite values

Recursion

- datatype as argument
- peel off one constructor
- recursive call only on arguments of the constructor

Corecursion

- codatatype as result
- produce one constructor
- corecursive call only in arguments to the constructor

```
∞ = eSuc ∞
```

Computing with infinite values

Computing with codatatypes is pattern matching on results

Definition (Productivity)

We can inspect arbitrary finite amounts of output in finitely many steps.

Am I productive?

```
s = 0 ## s
```

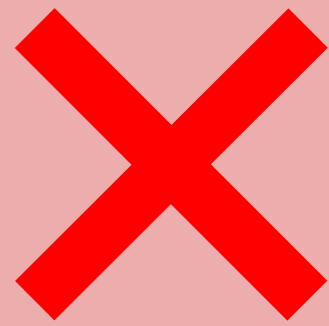
```
s = 0 ## s
```

primitive corecusion

```
$ ghci
Prelude> s = 0 : s
Prelude> take 5 s
[0,0,0,0,0]
Prelude>
```

```
s = 0 ## stl s
```

```
s = 0 ## stl s
```



`stl` evil

```
$ ghci
Prelude> s = 0 : tail s
Prelude> take 5 s
[0^CInterrupted
Prelude>
```

```
s = 0 ## 1 ## s
```

```
s = 0 ## 1 ## s
```

corecursion up to constructors

```
eo s = shd s ## eo (stl (stl s))
```

```
eo s = shd s ## eo (stl (stl s))
```

primitive corecusion

```
s = 0 ## 1 ## eo s
```

```
s = 0 ## 1 ## eo s
```

eo evil

$$s \oplus t = (shd\ s + shd\ t) \mathbin{\#\#} (stl\ s \oplus stl\ t)$$

$$s \oplus t = (\mathrm{shd}\ s + \mathrm{shd}\ t) \mathbin{\#\#} (\mathrm{stl}\ s \oplus \mathrm{stl}\ t)$$

primitive corecusion

$$s \otimes t = (shd\ s * shd\ t) \mathbin{\#\#} (stl\ s \otimes t \oplus s \otimes stl\ t)$$

$$s \otimes t = (\mathrm{shd}\ s * \mathrm{shd}\ t) \mathbin{\#\#} (\mathrm{stl}\ s \otimes t \oplus s \otimes \mathrm{stl}\ t)$$

corecursion up to $\oplus$

$$(\sigma \otimes \tau)(n) = \sum_{k=0}^{n} \binom{n}{k} \times \sigma(n-k) \times \tau(k)$$

The standard definition

$$s = 0 \mathbin{\#\#} ((1 \mathbin{\#\#} s) \oplus s)$$

$$s = 0 \mathbin{\#\#} ((1 \mathbin{\#\#} s) \oplus s)$$

corecursion up-to constructors and $\oplus$

$$s = (0 \mathbin{\#\#} 1 \mathbin{\#\#} s) \oplus (0 \mathbin{\#\#} s)$$

$$s = (0 \mathbin{\#\#} 1 \mathbin{\#\#} s) \oplus (0 \mathbin{\#\#} s)$$

corecursion up to constructors and $\oplus$

$\oplus$ comes before the guard

$$s = (1 \mathbin{\#\#} s) \otimes (1 \mathbin{\#\#} s)$$

$$s = (1 \mathbin{\#\#} s) \otimes (1 \mathbin{\#\#} s)$$

corecursion up to constructors and $\otimes$

$\otimes$ comes before the guard

```
selfie s = shd s ## selfie (selfie (stl s) ⊕ selfie s)
```

```
selfie s = shd s ## selfie (selfie (stl s) ⊕ selfie s)
```

corecursion up to $\oplus$ and `selfie` [sic!]

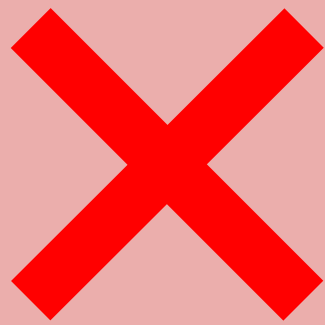
```
s m n =  if (m == 0 && n > 1) || gcd m n == 1
         then n ## s (m * n) (n + 1)
         else s m (n + 1)
```

```
s m n =  if (m == 0 && n > 1) || gcd m n == 1
         then n ## s (m * n) (n + 1)
         else s m (n + 1)
```

mixed recursion/primitive corecursion

```
s n =  if n > 0
       then stl (s (n - 1)) ⊕ (0 ## s (n + 1))
       else 1 ## s 1
```

```
s n =  if n > 0
       then stl (s (n - 1)) ⊕ (0 ## s (n + 1))
       else 1 ## s 1
```



`stl` really evil

Codatatypes and Corecursion

Codatatypes/Corecursion in Isabelle

Programming Language Examples

Codatatypes and Corecursion

Codatatypes/Corecursion in Isabelle

Programming Language Examples

Our Line of Work

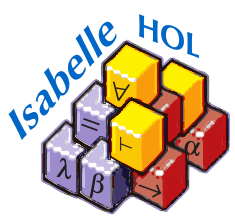
Foundational framework for

defining all the green stuff and more

Our Line of Work

Foundational framework for

defining all the green stuff and more

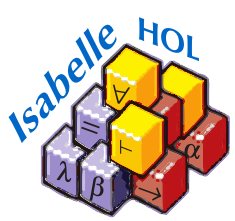
in an LCF-style proof assistant (

)

Our Line of Work

Foundational **framework** for

defining **all** the green stuff **and more**

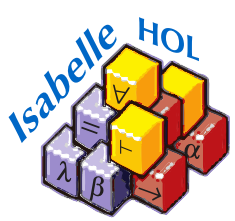
in an **LCF-style** proof assistant 

Burden on the **user**: prove { parametricity or termination } here and there

Our Line of Work

Foundational **framework** for

defining **all** the green stuff **and more**

in an **LCF-style** proof assistant (

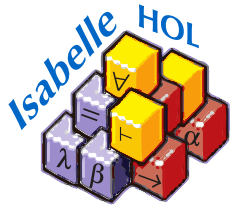
)

Burden on the **user**: prove { parametricity or termination } here and there

Most of the time: **automatic**

Guiding Principle

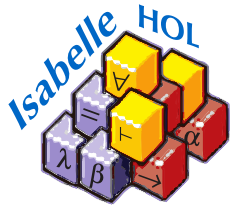
LCF Philosophy: Reduce everything to a small trusted kernel

Kernel of 

$\approx$
- simply typed lambda calculus $+$
- classical higher-order logic (axioms) $+$
- nonrecursive constant definition $+$
- nonrecursive type definition

Guiding Principle

LCF Philosophy: Reduce everything to a small trusted kernel

Kernel of 

$\approx$
- simply typed lambda calculus $+$
- classical higher-order logic (axioms) $+$
- nonrecursive constant definition $+$
- nonrecursive type definition

Our agenda make Isabelle/HOL a (co)recursion-friendly environment

LICS'12 ITP'14 IJCAR'14 ESOP'15 ICFP'15 ESOP'17 LICS'17

Related Work

Guarded Corecursion

- FRP (Krishnaswami & Benton, …)	type system
- clocks (Atkey & McBride)	type system
- guards (Clouston et al.)	type system
- abstract GSOS (Milius et al.)	category theory
- companions (Pous & Rot)	category theory (“up-to”)

Related Work

Guarded Corecursion / Proof Assistants

-	FRP (Krishnaswami & Benton, …)	type system
-	clocks (Atkey & McBride)	type system
-	guards (Clouston et al.)	type system
-	abstract GSOS (Milius et al.)	category theory
-	companions (Pous & Rot)	category theory (“up-to”)
Coq, Lean	constructor guarded corecursion	built-in
Agda	copatterns + sized types	built-in + type system
	up-to techniques, Thu@POPL 	
Dafny	mixed recursion/corecursion	built-in

Related Work

Guarded Corecursion / Proof Assistants

-	FRP (Krishnaswami & Benton, …)	type system
-	clocks (Atkey & McBride)	type system
-	guards (Clouston et al.)	type system
-	abstract GSOS (Milius et al.)	category theory
-	companions (Pous & Rot)	category theory (“up-to”)
Coq, Lean	constructor guarded corecursion	built-in
Agda	copatterns + sized types	built-in + type system
	up-to techniques, Thu@POPL 	
Dafny	mixed recursion/corecursion	built-in
Isabelle’	corecursion up-to *friendly* operations mixed with recursion	smart corecursor + wellfounded recursion

Primitive Corecursor

`codatatype` *Stream* = *Int ## Stream*

Primitive Corecursor

$\texttt{codatatype}\ \mathit{Stream} = \mathit{Int}\ \#\#\ \mathit{Stream}$

– $\mathit{Stream} \mathrel{\tilde{=}} \texttt{gfp}\,(\mathit{Int} \times -)$

– $\mathrm{corec}^{\mathrm{P}} : (A \to \mathit{Int} \times A) \to A \to \mathit{Stream}$

Primitive Corecursor

$\texttt{codatatype}\ \mathit{Stream} = \mathit{Int}\ \#\#\ \mathit{Stream}$

– $\mathit{Stream} \mathrel{\tilde{=}} \texttt{gfp}\,(\mathit{Int} \times -)$

– $\mathrm{corec}^{\mathrm{P}} : (A \to \mathit{Int} \times A) \to A \to \mathit{Stream}$

$\texttt{primcorec}\ s \oplus t = \quad (\mathrm{shd}\ s + \mathrm{shd}\ t)\,\#\#\,(\mathrm{stl}\ s \oplus \mathrm{stl}\ t)$

Primitive Corecursor

$\texttt{codatatype}\ \mathit{Stream} = \mathit{Int}\ \#\#\ \mathit{Stream}$

– $\mathit{Stream} \mathrel{\tilde{=}} \texttt{gfp}\ (\mathit{Int} \times -)$

– $\mathsf{corec}^{\mathsf{P}} : (A \to \mathit{Int} \times A) \to A \to \mathit{Stream}$

$\texttt{primcorec}\ s \oplus t = \quad (\mathsf{shd}\ s + \mathsf{shd}\ t) \#\# (\mathsf{stl}\ s \oplus \mathsf{stl}\ t)$

– $s \oplus t = \mathsf{corec}^{\mathsf{P}}\ (\lambda(s,t).\ ((\mathsf{shd}\ s + \mathsf{shd}\ t)\ ,\ (\mathsf{stl}\ s\ \ ,\ \ \mathsf{stl}\ t)))\ (s,t)$

Primitive Corecursor

`codatatype` $C = \cdots$

– $C \mathrel{\tilde{=}} \texttt{gfp}\ F$

– $\text{corec}^{\text{P}} : (A \to F\ A) \to A \to C$

`primcorec` $f\ \overline{x} = \quad \cdots$

– $f\ \overline{x} = \text{corec}^{\text{P}}\ (\lambda(\overline{x}).\ \cdots)\ (\overline{x})$

(Assuming F is a bounded natural functor)

Smart Corecursor

$$\mathrm{corec}^{P} : (A \to F\ A) \to A \to C$$

Smart Corecursor

$\mathrm{corec}^{P} : (A \to F\ A) \to A \to C$

$\mathrm{corec}_0^{S} : (A \to \blacksquare\ (F\ (\blacksquare\ A))) \to A \to C$

Smart Corecursor

$\mathrm{corec}^{P} : (A \to F\ A) \to A \to C$

$\mathrm{corec}_0^{S} : (A \to \blacksquare\ (F\ (\blacksquare\ A))) \to A \to C$

$\mathrm{corec}_1^{S} : (A \to$  $(F\ ($  $A))) \to A \to C$

Smart Corecursor

$\mathsf{corec}^{P} : (A \to F\ A) \to A \to C$

$\mathsf{corec}_0^{S} : (A \to \blacksquare\ (F\ (\blacksquare\ A))) \to A \to C$

$\mathsf{corec}_1^{S} : (A \to \oplus\ (F\ (\oplus\ A))) \to A \to C$

$$\mathtt{corec}\ s \otimes t = (\mathsf{shd}\ s * \mathsf{shd}\ t) \#\# (\mathsf{stl}\ s \otimes t\ \oplus\ s \otimes \mathsf{stl}\ t)$$

– $s \otimes t = \mathsf{corec}_1^{S}\ (\lambda(s,t).$
$\eta\left((\mathsf{shd}\ s * \mathsf{shd}\ t)\ ,\ \eta(\mathsf{stl}\ s,\ t)\ \overline{\oplus}\ \eta(s,\ \mathsf{stl}\ t)\right))\ (s,t)$

– $\overline{\oplus} : \blacksquare\ A \to \blacksquare\ A \to \blacksquare\ A$

– $\eta : A \to \blacksquare\ A$

Smart Corecursor

$\mathrm{corec}^{P} : (A \to F\ A) \to A \to C$

$\mathrm{corec}_0^{S} : (A \to \blacksquare\ (F\ (\blacksquare\ A))) \to A \to C$

$\mathrm{corec}_1^{S} : (A \to \oplus\ (F\ (\oplus\ A))) \to A \to C$

$\mathrm{corec}_2^{S} : (A \to \circledast\ (F\ (\circledast\ A))) \to A \to C$

$\mathrm{corec}\ s \otimes t = (\mathrm{shd}\ s * \mathrm{shd}\ t) \#\# (\mathrm{stl}\ s \otimes t\ \oplus\ s \otimes \mathrm{stl}\ t)$

– $s \otimes t = \mathrm{corec}_1^{S}\ (\lambda(s,t).$
$\eta\left((\mathrm{shd}\ s * \mathrm{shd}\ t)\ ,\ \eta(\mathrm{stl}\ s,\ t)\,\overline{\oplus}\,\eta(s,\ \mathrm{stl}\ t)\right))\ (s,t)$

– $\overline{\oplus} : \oplus\ A \to \oplus\ A \to \oplus\ A$

– $\eta : A \to \oplus\ A$

$\otimes : C \to C \to C$ has to be friendly

A friendly function can consume
one constructor to produce
at least one constructor.

$\otimes : C \to C \to C$ has to be friendly

$\exists$ parametric $\rho_{\otimes} : (A \times F\ A) \to (A \times F\ A) \to F\ (\otimes\ A)$ s.t.

$$s \otimes t = \cdots\ (\rho_{\otimes}\ (\cdots\ (s, t)))$$

$\otimes : C \to C \to C$ has to be friendly

$\exists$ parametric $\rho_{\otimes} : (A \times F\ A) \to (A \times F\ A) \to F\ (\circledast\ A)$ s.t.

$$s \otimes t = \cdots\ (\rho_{\otimes}\ (\cdots\ (s, t)))$$

$$\rho_{\otimes} : (A \times (Int \times A)) \to (A \times (Int \times A)) \to (Int \times \circledast\ A)$$

$$\rho_{\otimes}\ (s, hs, ts)\ (t, ht, tt) = (hs * ht, \eta\ ts\ \overline{\otimes}\ \eta\ t\ \overline{\oplus}\ \eta\ s\ \overline{\otimes}\ \eta\ tt)$$

Isabelle Demo

Codatatypes and Corecursion

Codatatypes/Corecursion in Isabelle

Programming Language Examples

Streams and Infinite Trees

```
codatatype 'a stream =
  SCons (shd: 'a)
        (stl: 'a stream)
```

```
up = 0 ## smap (λx. x + 1) up
```

[0,1,2,3,4,5,6,...]

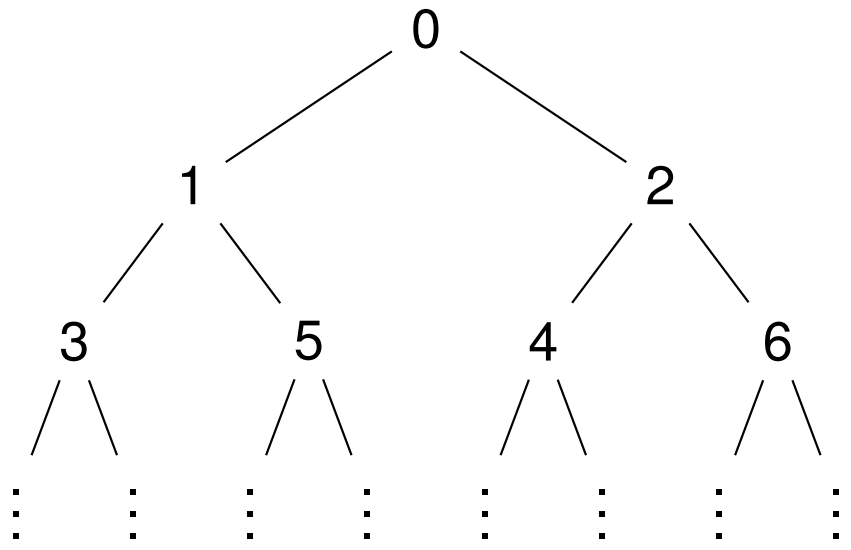
Streams and Infinite Trees

```
codatatype 'a stream =
  SCons (shd: 'a)
        (stl: 'a stream)
```

```
up = 0 ## smap (λx. x + 1) up
```

[0,1,2,3,4,5,6,...]

```
codatatype 'a tree =
  Node (left: 'a tree)
       (root: 'a)
       (right: 'a tree)
```



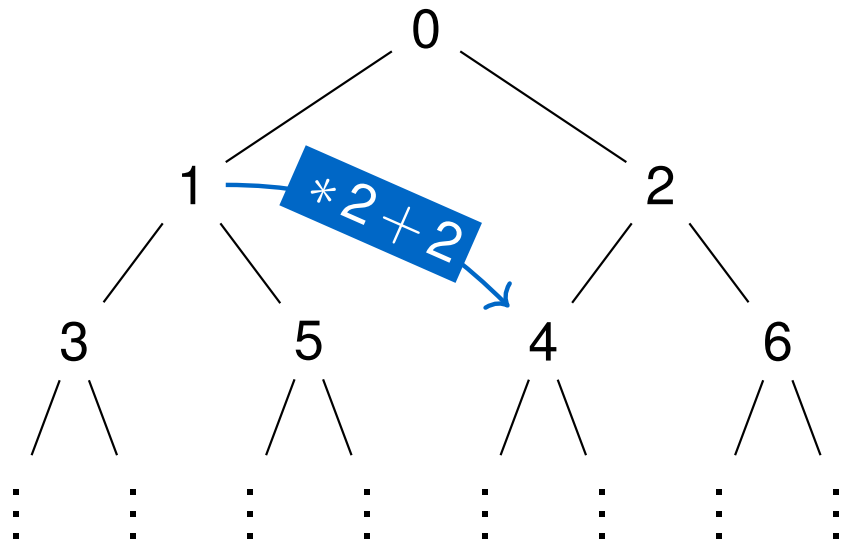
Streams and Infinite Trees

```
codatatype 'a stream =
  SCons (shd: 'a)
        (stl: 'a stream)
```

```
up = 0 ## smap (λx. x + 1) up
```

[0,1,2,3,4,5,6,...]

```
codatatype 'a tree =
  Node (left: 'a tree)
       (root: 'a)
       (right: 'a tree)
```



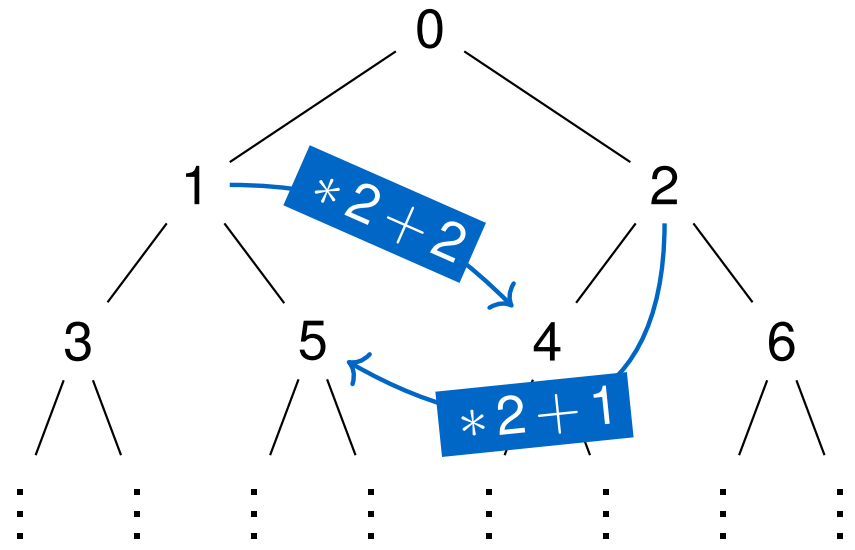
Streams and Infinite Trees

```
codatatype 'a stream =
  SCons (shd: 'a)
        (stl: 'a stream)
```

```
up = 0 ## smap (λx. x + 1) up
```

[0,1,2,3,4,5,6,...]

```
codatatype 'a tree =
  Node (left: 'a tree)
       (root: 'a)
       (right: 'a tree)
```




```
tnum = Node (tmap (λx. 2 * x + 1) tnum)
            0 (tmap (λx. 2 * x + 2) tnum)
```

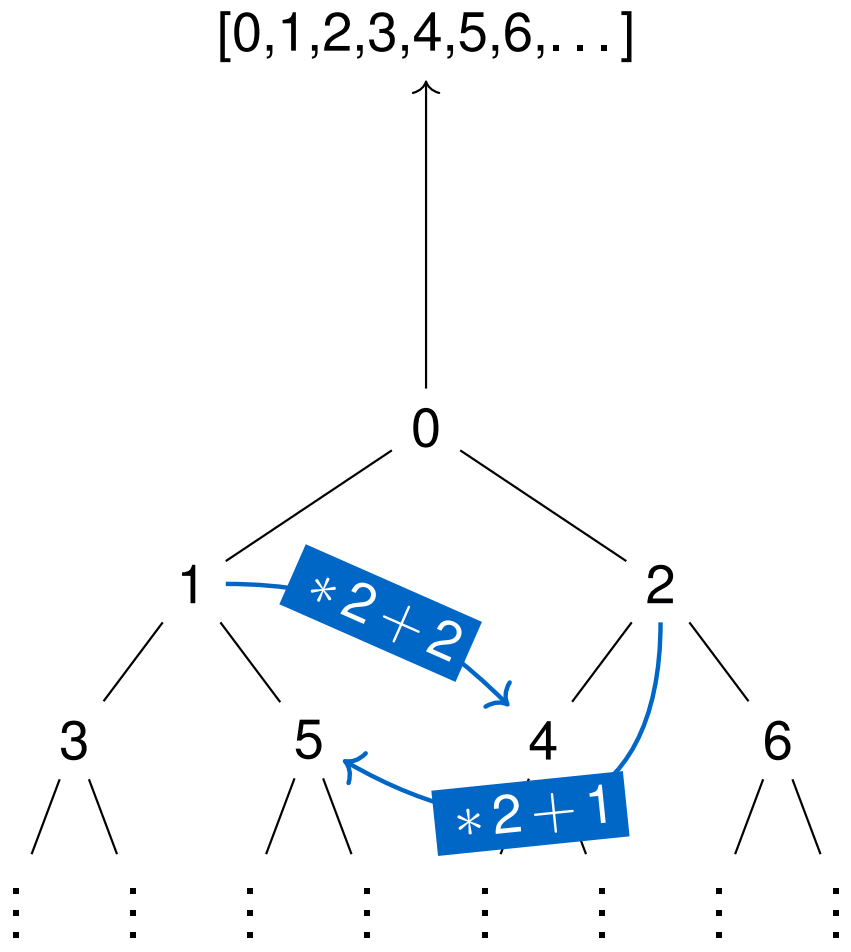
Streams and Infinite Trees

```
codatatype 'a stream =
  SCons (shd: 'a)
        (stl: 'a stream)
```

stream_of

```
codatatype 'a tree =
  Node (left: 'a tree)
       (root: 'a)
       (right: 'a tree)
```

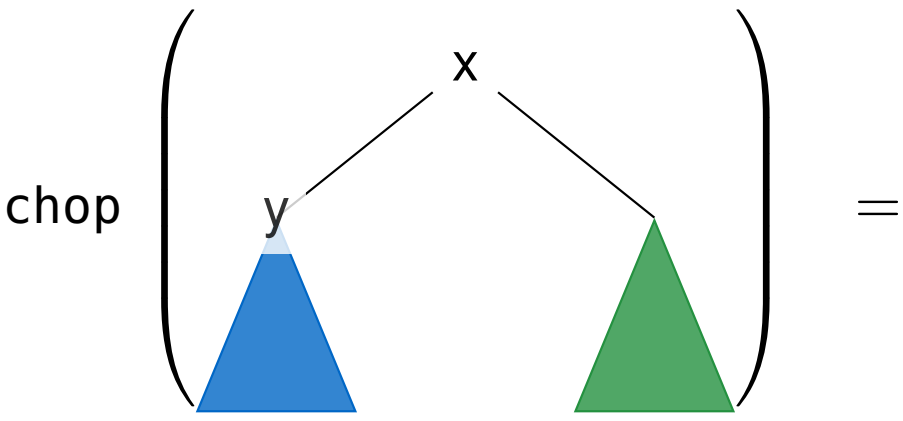
```
up = 0 ## smap (λx. x + 1) up
```




```
tnum = Node (tmap (λx. 2 * x + 1) tnum)
            0 (tmap (λx. 2 * x + 2) tnum)
```

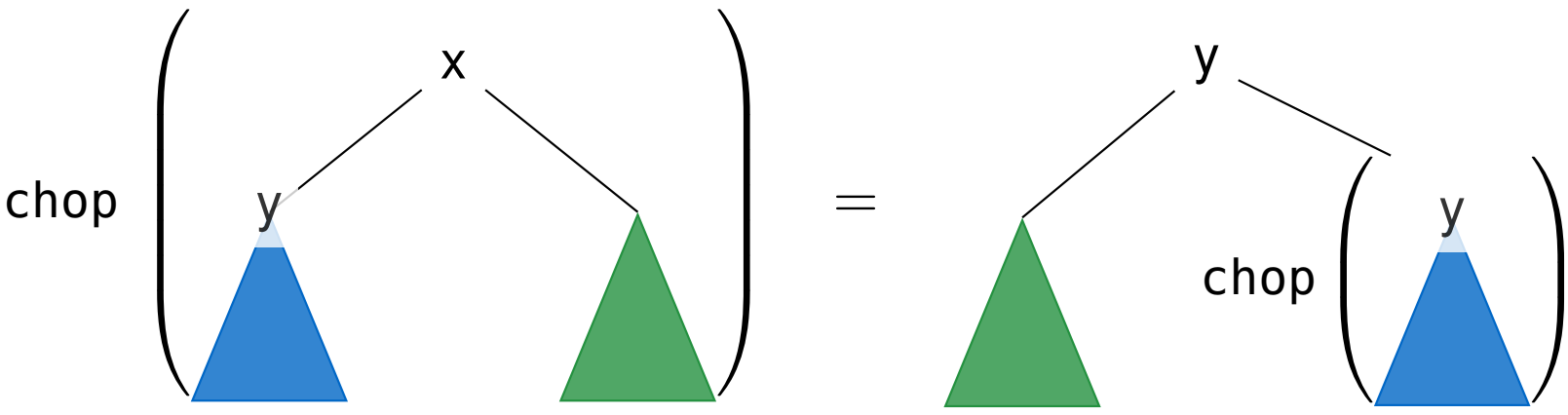
Tree chopping

Remove the root of a tree:



Tree chopping

Remove the root of a tree:



Coinductive traces for a WHILE langauge

Syntax

com ::= SKIP
| ATOM atom
| com ;; com
| IF test com com
| WHILE test com com

Semantic domain

step: state update or test result

trace: [step* | state] or step$^\infty$

```
codatatype trace
  = Final state
  | Step step trace
```

process: state $\Rightarrow$ trace

Coinductive traces for a WHILE langauge

Syntax

com ::= SKIP
| ATOM atom
| com ;; com
| IF test com com
| WHILE test com com

Semantic domain

step: state update or test result

trace: [step* | state] or step$^\infty$

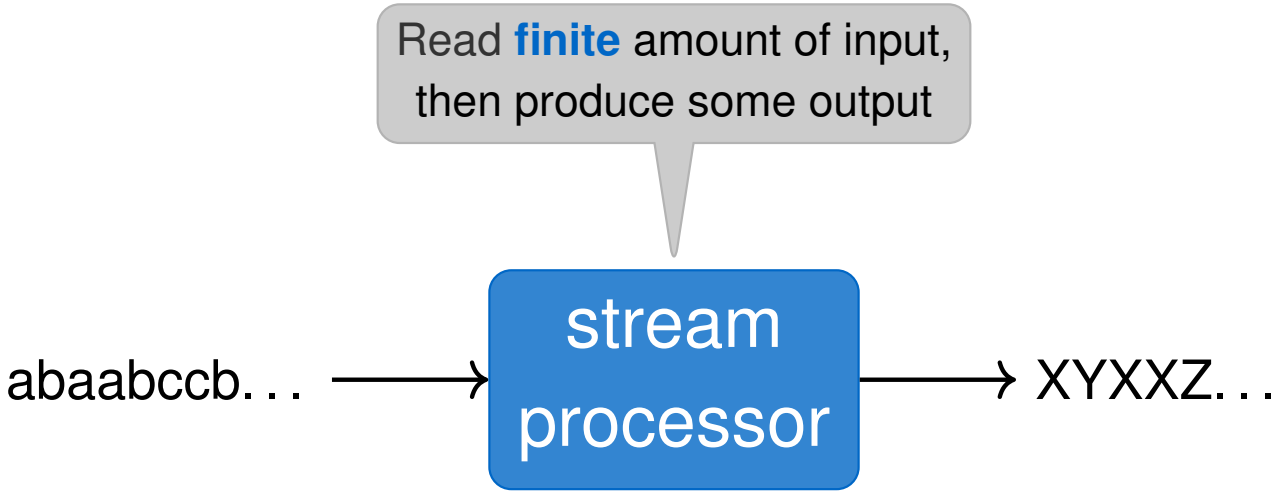
```
codatatype trace
  = Final state
  | Step step trace
```

process: state $\Rightarrow$ trace

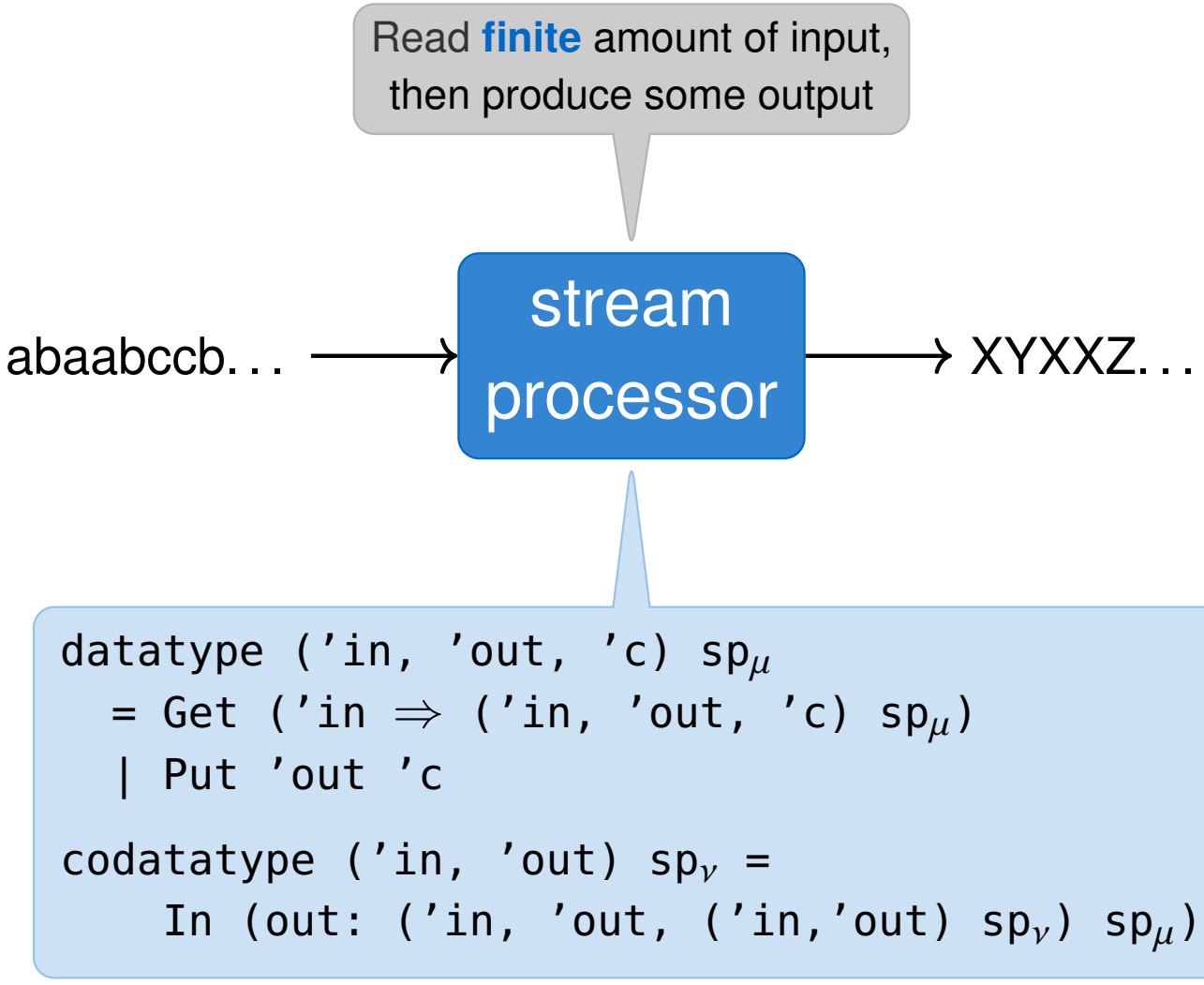
Trace semantics

$[\![\_]\!]$ :: com $\Rightarrow$ process
$[\![$ SKIP $]\!]$ = (λs. [| s])
$[\![$ ATOM atom $]\!]$ = (λs. [Update s | evalA atom s])
$\vdots$

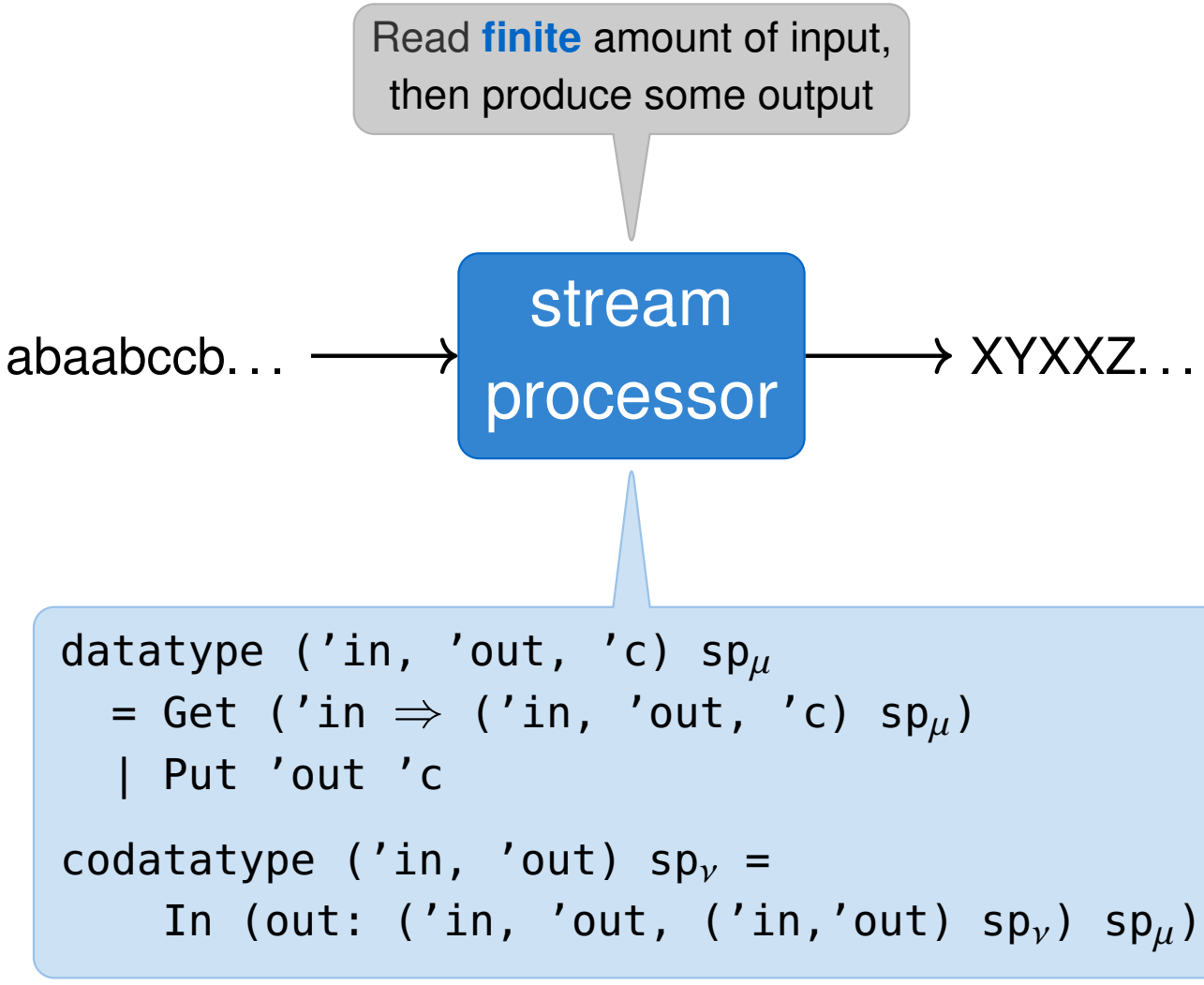
Stream processors



Stream processors



Stream processors

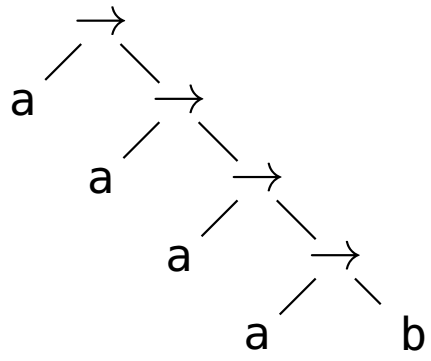


Calculator example by Hur et al. [POPL 2013]

Equirecursive types

Semantics: Infinite types

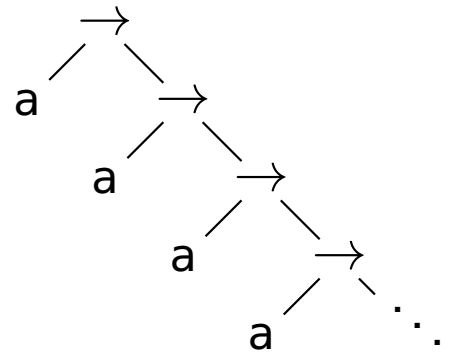
```
codatatype type
  = TVar var
  | type → type
```



Equirecursive types

Semantics: Infinite types

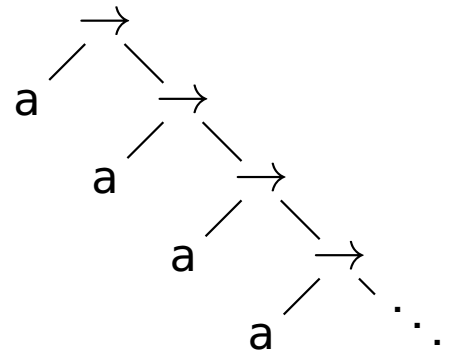
```
codatatype type
  = TVar var
  | type → type
```



Equirecursive types

Semantics: Infinite types

```
codatatype type
  = TVar var
  | type → type
```



Syntax: Recursive types

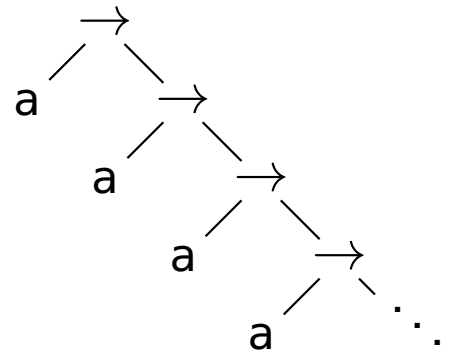
```
datatype ty
  = Var var
  | Arrow ty ty
  | Mu var ty
```

```
Mu X. Arrow (Var a) (Var X)
```

Equirecursive types

Semantics: Infinite types

```
codatatype type
  = TVar var
  | type → type
```



$$[\![\_]\!] :: \text{ty} \Rightarrow \text{type}$$

$$[\![\text{Mu X T}]\!] = \left[\text{X} \mapsto [\![\text{Mu X T}]\!]\right] \cdot [\![\text{T}]\!]$$

Syntax: Recursive types

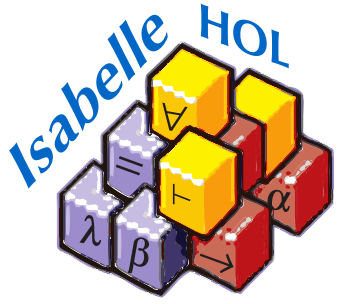
```
datatype ty
  = Var var
  | Arrow ty ty
  | Mu var ty
```

```
Mu X. Arrow (Var a) (Var X)
```

Outlook

- **Stern-Brocot tree of rational numbers**
  `http://www.isa-afp.org/entries/Stern_Brocot.shtml`
- **Knuth-Morris-Pratt string matching**
  [ESOP 2017]
- **Regular languages via Brzozowski derivatives**
  [FSCD 2016]
- **Probabilistic reactive programming**
  CryptHOL
  TLS in Isabelle

Programming and Reasoning with Infinite Data in



Thank You