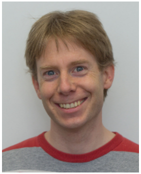
# Certified Functional (Co)programming with Isabelle/HOL


Jasmin Blanchette — VU Vrije Universiteit Amsterdam

Andreas Lochbihler — ETH zürich

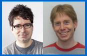
Andrei Popescu — Middlesex University London

Dmitriy Traytel — ETH zürich

Partly based on material by Tobias Nipkow

Preliminaries 

Programming 

Coprogramming 

Advanced Coprogramming 

## Preliminaries

Applications

Higher-Order Logic

Proving

## Programming

## Coprogramming

## Advanced Coprogramming

# Big proofs about programs

(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

Lammich, Popescu et al.

### 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 2WL

Fleury, Blanchette et al.

HOL = Higher-Order Logic

HOL = Higher-Order Logic

HOL = Functional Programming + Logic

HOL = Higher-Order Logic

HOL = Functional Programming + Logic

HOL has

- (co)datatypes
- (co)recursive functions
- logical operators

HOL = Higher-Order Logic

HOL = Functional Programming + Logic

HOL has

- (co)datatypes
- (co)recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too

HOL = Higher-Order Logic

HOL = Functional Programming + Logic

HOL has

- (co)datatypes
- (co)recursive functions
- logical operators

HOL is a programming language!

Higher-order = functions are values, too

HOL formulas:

- Equations: *term* $=$ *term*,
  e.g. $1 + 2 = 4$
- Also: $\wedge$, $\vee$, $\longrightarrow$, $\forall$, $\exists$, $\ldots$

# Types

## Basic syntax

| | | | |
|---|---|---|---|
| $\tau$ | ::= | $(\tau)$ | |
| | $\mid$ | `bool` $\mid$ `nat` $\mid$ `int` $\mid \cdots$ | base types |
| | $\mid$ | `'a` $\mid$ `'b` $\mid \cdots$ | type variables |
| | $\mid$ | $\tau \Rightarrow \tau$ | functions (ASCII: =>) |
| | $\mid$ | $\tau \times \tau$ | pairs (ASCII: $*$) |
| | $\mid$ | $\tau$ `list` | lists |
| | $\mid$ | $\tau$ `set` | sets |
| | $\mid$ | ... | user-defined types |

# Terms

## Basic syntax

| | | | |
|---|---|---|---|
| $t$ | $::=$ | $(t)$ | |
| | $\mid$ | $a$ | constant or variable (identifier) |
| | $\mid$ | $t\ t$ | function application |
| | $\mid$ | $\lambda x.\ t$ | function abstraction |
| | $\mid$ | $\ldots$ | lots of syntactic sugar |

Examples:
```
f (g x) y
h (%x. f (g x))
```

Terms must be well-typed

(the argument of every function call must be of the right type)

**Terms must be well-typed**

(the argument of every function call must be of the right type)

Notation:
$t :: \tau$ means "$t$ is a well-typed term of type $\tau$".

$$\frac{t :: \tau_1 \Rightarrow \tau_2 \qquad u :: \tau_1}{t\ u :: \tau_2}$$

# Type inference

Isabelle automatically computes the type of each variable in a term.

In the presence of *overloaded* functions (functions with multiple types) this is not always possible.

Users can help with type annotations inside the term.
Example: `f (x::nat)`

# Currying

Thou shalt curry thy functions

- Curried: `f ::` $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- Tupled: `f' ::` $\tau_1 \times \tau_2 \Rightarrow \tau$

# Currying

Thou shalt curry thy functions

- Curried: `f ::` $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau$
- Tupled: `f' ::` $\tau_1 \times \tau_2 \Rightarrow \tau$

Advantage:

Currying allows *partial application*

`f` $a_1$ where $a_1$ `::` $\tau_1$

# Formulas

Metalogic (Pure):

- Type `prop`
- Constants:

$$\bigwedge :: (\texttt{'a} \Rightarrow \texttt{prop}) \Rightarrow \texttt{prop}$$

$$\Longrightarrow :: \texttt{prop} \Rightarrow \texttt{prop} \Rightarrow \texttt{prop}$$

$$\equiv :: \texttt{'a} \Rightarrow \texttt{'a} \Rightarrow \texttt{prop}$$

# Formulas

Metalogic (Pure):

- Type `prop`
- Constants:

  $\bigwedge :: (\texttt{'a} \Rightarrow \texttt{prop}) \Rightarrow \texttt{prop}$

  $\Longrightarrow :: \texttt{prop} \Rightarrow \texttt{prop} \Rightarrow \texttt{prop}$

  $\equiv :: \texttt{'a} \Rightarrow \texttt{'a} \Rightarrow \texttt{prop}$

Object logic (HOL):

- Type `bool`
- Constants:

  $\texttt{Trueprop} :: \texttt{bool} \Rightarrow \texttt{prop}$ (implicit)

  $\forall, \exists :: (\texttt{'a} \Rightarrow \texttt{bool}) \Rightarrow \texttt{bool}$

  $\longrightarrow, \wedge, \vee, \ldots :: \texttt{bool} \Rightarrow \texttt{bool} \Rightarrow \texttt{bool}$

  $= :: \texttt{'a} \Rightarrow \texttt{'a} \Rightarrow \texttt{bool}$

# Syntactic sugar

- *Infix:* `+`, `-`, `*`, `#`, `@`, ...
- *Mixfix:* `if _ then _ else _`, `case _ of`, ...
- *Binders:* $\bigwedge$`_. _`, `∀_. _`, `∃_. _`, ...

# Syntactic sugar

- *Infix:* `+`, `-`, `*`, `#`, `@`, ...
- *Mixfix:* `if _ then _ else _`, `case _ of`, ...
- *Binders:* $\bigwedge$`_. _`, $\forall$`_. _`, $\exists$`_. _`, ...

Prefix binds more strongly than infix:

`f x + y` $\equiv$ `(f x) + y` $\not\equiv$ `f (x + y)`

Enclose `if` and `case` in parentheses:

`(if _ then _ else _)`

# Theory = Isabelle Module

# Theory = Isabelle Module

## Syntax

**theory** *MyTh*
**imports** $T_1 \ldots T_n$
**begin**
(definitions, theorems, proofs, ...)$^*$
**end**

# Theory = Isabelle Module

## Syntax

```
theory MyTh
imports T1 ... Tn
begin
(definitions, theorems, proofs, ...)*
end
```

*MyTh*: name of theory. Must live in file *MyTh*`.thy`

$T_i$: names of *imported* theories. Import transitive.

Typically: `imports Main`

# Concrete syntax

In `.thy` files:

Types, terms and formulas need to be inclosed in double quotes (")

# Concrete syntax

In `.thy` files:

Types, terms and formulas need to be inclosed in double quotes (`"`)

except for single identifiers.

# Concrete syntax

In `.thy` files:

Types, terms and formulas need to be inclosed in double quotes (`"`)

except for single identifiers.

Double quotes are not always shown on slides.

# Isabelle/jEdit

- Based on the jEdit editor
- Processes Isabelle text automatically
  when editing `.thy` files (like modern Java IDEs)

- Bottom panels: Output, Query (Find Theorems), ...
- Side panels: State, Theories, ...

# The proof state

1. $\bigwedge x_1 \dots x_m.\ A_1 \Longrightarrow \dots \Longrightarrow A_n \Longrightarrow C$

| | |
|---|---|
| $x_1, \dots, x_m$ | fixed local variables |
| $A_1, \dots, A_n$ | local assumptions |
| $C$ | actual (sub)goal |

# Apply scripts

General schema:

```
lemma name: "..."
apply (...)
  ⋮
apply (...)
apply (...)
done
```

```
lemma name: "..."
apply (...)
  ⋮
apply (...)
by (...)
```

# Apply scripts

General schema:

```
lemma name: "..."
apply (...)
  ⋮
apply (...)
apply (...)
done
```

```
lemma name: "..."
apply (...)
  ⋮
apply (...)
by (...)
```

If the lemma is suitable as a simplification rule:

```
lemma name[simp]: "..."
```

# Delayed gratification

The command `oops` gives up the current proof attempt.

The command `sorry` “completes” any proof.
It makes top-down development possible:

*Assume lemma first, prove it later.*

# Isar Proofs

Apply script = assembly language program

# Isar Proofs

Apply script = assembly language program

Isar proof = structured program with comments

# Isar Proofs

Apply script = assembly language program

Isar proof = structured program with comments

But `apply` still useful for proof exploration.

# A typical Isar proof

A proof of $\varphi_0 \Longrightarrow \varphi_{n+1}$:

```
proof
  assume φ0
  have φ1
    by simp
   ⋮
  have φn
    by blast
  show φn+1
    by ...
qed
```

# Isar core syntax

$$
\begin{array}{lcl}
\textit{proof} & = & \texttt{proof}\ [\textit{method}]\ \ \textit{step}^*\ \ \texttt{qed} \\
 & | & \texttt{by}\ \textit{method} \\[1em]
\textit{method} & = & \texttt{(simp ...)} \mid \texttt{(blast ...)} \mid \texttt{(induction ...)} \mid \cdots \\[1em]
\textit{step} & = & \texttt{fix}\ \textit{variables} \qquad (\bigwedge) \\
 & | & \texttt{assume}\ \textit{prop} \qquad (\Longrightarrow) \\
 & | & [\texttt{from}\ \textit{fact}^+]\ (\texttt{have} \mid \texttt{show})\ \textit{prop}\ \ \textit{proof} \\[1em]
\textit{prop} & = & [\textit{name}\texttt{:}]\ \texttt{"}\textit{formula}\texttt{"} \\[1em]
\textit{fact} & = & \textit{name} \mid \cdots
\end{array}
$$

# Example: Cantor's theorem

```
lemma "¬ surj (f :: 'a ⇒ 'a set)"
proof
  assume a: "surj f"
  from a have b: "∀A. ∃a. A = f a"
    by (simp add: surj_def)
  from b have c: "∃a. {x. x ∉ f x} = f a"
    by blast
  from c show False
    by blast
qed
```

# Abbreviations

| | | |
|---|---|---|
| *this* | = | the previous proposition proved or assumed |
| `then` | = | `from` *this* |

# `using` and `with`

(`have` | `show`) *prop* `using` *facts*

# `using` and `with`

(`have` | `show`) *prop* `using` *facts*

=

`from` *facts* (`have` | `show`) *prop*

# `using` and `with`

$$(\texttt{have} \mid \texttt{show})\ \textit{prop}\ \texttt{using}\ \textit{facts}$$

$$=$$

$$\texttt{from}\ \textit{facts}\ (\texttt{have} \mid \texttt{show})\ \textit{prop}$$

$$\texttt{with}\ \textit{facts}$$

$$=$$

$$\texttt{from}\ \textit{facts}\ \textit{this}$$

# Structured lemma statement

```
lemma
  fixes f :: "'a ⇒ 'a set"
  assumes s: "surj f"
  shows False
proof -
  have "∃a. {x. x ∉ f x} = f a"
    using s by (auto simp: surj_def)
  then show False
    by blast
qed
```

# Structured lemma statement

```
lemma
  fixes f :: "'a ⇒ 'a set"
  assumes s: "surj f"
  shows False
proof -
  have "∃a. {x. x ∉ f x} = f a"
    using s by (auto simp: surj_def)
  then show False
    by blast
qed
```

*Proves* $surj\ f \Longrightarrow False$

*but* `surj f` *becomes local fact s in proof.*

# Structured lemma statements

*fixes* $x :: \tau_1$ *and* $y :: \tau_2 \ldots$
*assumes* $a$: $P$ *and* $b$: $Q \ldots$
*shows* $R$

# Structured lemma statements

*fixes* $x :: \tau_1$ *and* $y :: \tau_2 \ldots$
*assumes* $a$: $P$ *and* $b$: $Q \ldots$
*shows* $R$

- `fixes` and `assumes` sections optional
- `shows` optional if no `fixes` and `assumes`

# Case distinction

```
show "R"
proof cases
  assume "P"
  ...
  show "R" ...
next
  assume "¬ P"
  ...
  show "R" ...
qed
```

# Case distinction

```
show "R"
proof cases
  assume "P"
  ...
  show "R" ...
next
  assume "¬ P"
  ...
  show "R" ...
qed
```

```
have "P ∨ Q" ...
then show "R"
proof
  assume "P"
  ...
  show "R" ...
next
  assume "Q"
  ...
  show "R" ...
qed
```

# Contradiction

```
show "¬ P"
proof
  assume "P"
  ...
  show False ...
qed
```

# Contradiction

```
show "¬ P"
proof
  assume "P"
  ...
  show False ...
qed
```

```
show "P"
proof (rule ccontr)
  assume "¬ P"
  ...
  show False ...
qed
```

# $\longleftrightarrow$

```
show "P ⟷ Q"
proof
  assume "P"
  ...
  show "Q" ...
next
  assume "Q"
  ...
  show "P" ...
qed
```

# ∀ and ∃ introduction

```
show "∀x. P x"
proof
  fix x
  show "P x" ...
qed
```

# ∀ and ∃ introduction

```
show "∀x. P x"
proof
  fix x
  show "P x" ...
qed
```

```
show "∃x. P x"
proof
  ...
  show "P witness" ...
qed
```

Preliminaries

Programming


Datatypes

Recursion

Induction

Coprogramming

Advanced Coprogramming


# Natural numbers

```
datatype nat =
  0
| Suc nat
```

# Natural numbers

```
datatype nat =
  0
| Suc nat
```

This introduces:

- The type `nat`
- The constructors `0 :: nat` and `Suc :: nat ⇒ nat`
- A case combinator `case_nat`
- A (primitive) recursor constant `rec_nat`
- Various theorems about the above, including an induction rule

Numeral notations are also supported:

`3 = Suc (Suc (Suc 0)` is a theorem

# Lists

```
datatype 'a list =
  Nil
| Cons 'a "'a list"
```

# Lists

```
datatype 'a list =
  Nil
| Cons 'a "'a list"
```

More honest

```
datatype (set: 'a) list =
  Nil  ("[]")
| Cons 'a "'a list"  (infixr "#" 65)
for map: map
```

# Lists

```
datatype 'a list =
  Nil
| Cons 'a "'a list"
```

More honest

```
datatype (set: 'a) list =
  Nil  ("[]")
| Cons 'a "'a list"  (infixr "#" 65)
for map: map
```

This introduces:

- The type `'a list` and the constructors `Nil` and `Cons`
- A functorial action: `map :: ('a ⇒ 'b) ⇒ 'a list ⇒ 'b list`
- A natural transformation: `set :: 'a list ⇒ 'a set`
- A size function: `size :: 'a list ⇒ nat`
- etc.

# List notations

Empty list: `[]`

Cons: `x # xs`

Enumeration: `[`$x_1$`, `$x_2$`, ..., `$x_n$`]`

Head: `hd (x # xs) = x`

Tail: `tl (x # xs) = xs`    `tl [] = []`

Case: `(case xs of [] ⇒ ... | y # ys ⇒ ...)`

# Primitive recursion

Definition using `primrec` or `fun`:

```
primrec append :: 'a list ⇒ 'a list ⇒ 'a list (infixr "@" 65)
where
  [] @ ys = ys
| (x # xs) @ ys = x # (xs @ ys)
```

# Primitive recursion

Definition using `primrec` or `fun`:

```
primrec append :: 'a list ⇒ 'a list ⇒ 'a list (infixr "@" 65)
where
  [] @ ys = ys
| (x # xs) @ ys = x # (xs @ ys)
```

Code export:

```
export_code append in Haskell
```

# Primitive recursion

Definition using `primrec` or `fun`:

```
primrec append :: 'a list ⇒ 'a list ⇒ 'a list (infixr "@" 65)
where
  [] @ ys = ys
| (x # xs) @ ys = x # (xs @ ys)
```

Code export:

```
export_code append in Haskell
```

Symbolic evaluation in Isabelle:

```
value "[1,2,3] @ [4,5,6] :: int list"
```

# Structural induction

We want to prove `xs @ [] = xs`.

# Structural induction

We want to prove `xs @ [] = xs`.

**Structural induction rule** (`thm list.induct`)

```
?P [] ⟹                                    (base case)
(⋀x xs. ?P xs ⟹ ?P (x # xs)) ⟹            (induction step)
?P ?list
```

# Structural induction

We want to prove `xs @ [] = xs`.

**Structural induction rule** (`thm list.induct`)

```
?P [] ⟹                                          (base case)
(⋀x xs. ?P xs ⟹ ?P (x # xs)) ⟹                   (induction step)
?P ?list
```

Base case:

`[] @ [] = []` (by definition of `@`)

# Structural induction

We want to prove `xs @ [] = xs`.

**Structural induction rule** (thm `list.induct`)

$?P\ []\ \Longrightarrow$ (base case)

$(\bigwedge x\ xs.\ ?P\ xs \Longrightarrow\ ?P\ (x\ \#\ xs)) \Longrightarrow$ (induction step)

$?P\ ?list$

**Base case:**

`[] @ [] = []` (by definition of @)

**Induction step:**

$(x\ \#\ xs)\ @\ [] = x\ \#\ (xs\ @\ [])$ (by definition of @)

$\quad = x\ \#\ xs$ (by induction hypothesis)

# List reversal

$$\texttt{[1,2,3,4]} \xrightarrow{\texttt{rev}} \texttt{[4,3,2,1]}$$

# List reversal

$$\texttt{[1,2,3,4]} \xrightarrow{\texttt{rev}} \texttt{[4,3,2,1]}$$

## Naive list reversal

```
primrec rev :: 'a list ⇒ 'a list where
  rev []        = []
| rev (x # xs) = rev xs @ [x]
```

# List reversal

$$\texttt{[1,2,3,4]} \xrightarrow{\texttt{rev}} \texttt{[4,3,2,1]}$$

## Naive list reversal

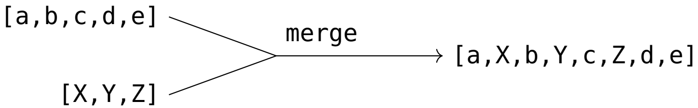
```
primrec rev :: 'a list ⇒ 'a list where
  rev []        = []
| rev (x # xs) = rev xs @ [x]
```

## Fast list reversal

```
primrec qrev :: 'a list ⇒ 'a list ⇒ 'a list where
  qrev []        ys = ys
| qrev (x # xs) ys = qrev xs (x # ys)
```

1. What is `rev (rev xs)`?
2. Are `rev` and `qrev` equivalent? What is the relationship?

# Well-founded recursion



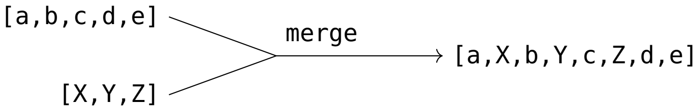

## Merge two lists

```
function merge :: 'a list ⇒ 'a list ⇒ 'a list where
  merge []       ys = ys
| merge (x # xs) ys = x # merge ys xs
```

Not primitively recursive! ⇝ We need a termination proof:

```
termination
proof(relation "measure (λ(xs, ys). size xs + size ys)")
qed simp_all
```

# Well-founded recursion




## Merge two lists

```
function merge :: 'a list ⇒ 'a list ⇒ 'a list where
  merge []       ys = ys
| merge (x # xs) ys = x # merge ys xs
```

Not primitively recursive! ⇝ We need a termination proof:

```
termination
proof(relation "measure (λ(xs, ys). size xs + size ys)")
qed simp_all
```

With proof automation:

```
termination by size_change
```

# Termination proofs yield induction rules

```
merge []       ys = ys
merge (x # xs) ys = x # merge ys xs
```

How can we prove `size (merge xs ys) = size xs + size ys`?

# Termination proofs yield induction rules

```
merge []       ys = ys
merge (x # xs) ys = x # merge ys xs
```

How can we prove `size (merge xs ys) = size xs + size ys`?

### Structural induction on `xs` does not work!

Induction hypothesis: `size (merge xs ys) = size xs + size ys`

```
size (merge (x # xs) ys)
= size (x # merge ys xs) = 1 + size (merge ys xs) = ...
```

# Termination proofs yield induction rules

```
merge []       ys = ys
merge (x # xs) ys = x # merge ys xs
```

How can we prove `size (merge xs ys) = size xs + size ys`?

## Structural induction on `xs` does not work!

Induction hypothesis: `size (merge xs ys) = size xs + size ys`

```
size (merge (x # xs) ys)
= size (x # merge ys xs) = 1 + size (merge ys xs) = ...
```

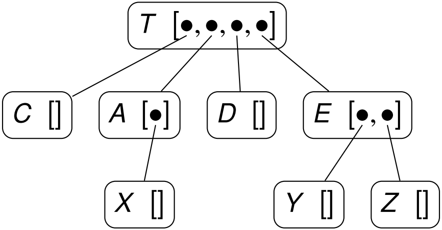
## Induction rule for `merge`

(⋀ys. ?P [] ys) ⟹ (1st equation)

(⋀x xs ys. ?P ys xs ⟹ ?P (x # xs) ys) ⟹ (2nd equation)

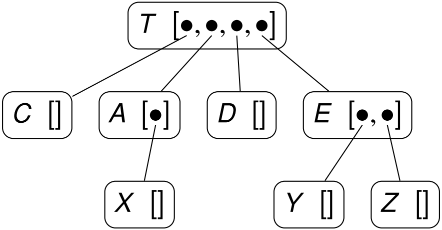
?P ?xs ?ys

# Finitely-branching Rose trees




```
datatype 'a rtree = Node 'a "'a rtree list"
```

# Finitely-branching Rose trees

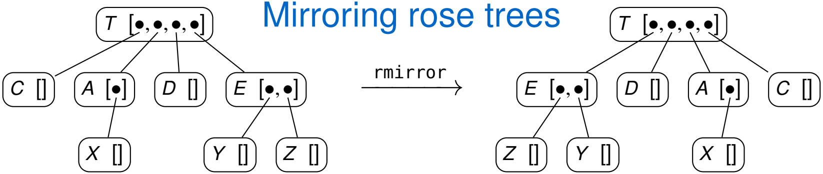



```
datatype 'a rtree = Node 'a "'a rtree list"
```

## Structural induction

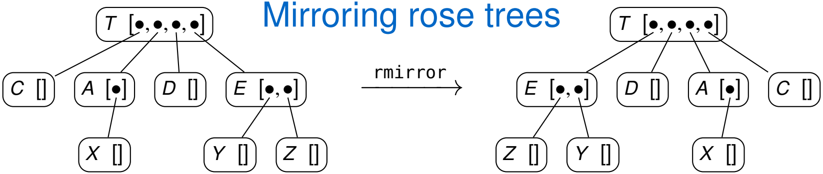
(⋀x ts. (⋀t. t ∈ set ts ⟹ ?P t) ⟹ ?P (Node x ts)) ⟹
?P ?tree

# Mirroring rose trees




```
primrec rmirror :: 'a tree ⇒ 'a tree where
  rmirror (Node x ts) = Node x (rev (map rmirror ts))
```

# Mirroring rose trees




```
primrec rmirror :: 'a tree ⇒ 'a tree where
  rmirror (Node x ts) = Node x (rev (map rmirror ts))
```

**Prove** `rmirror (rmirror t) = t` **by structural induction**

IH: `rmirror (rmirror t) = t` for all `t ∈ set ts`

```
rmirror (rmirror (Node x ts))
= rmirror (Node x (rev (map rmirror ts)))
= Node x (rev (map rmirror (rev (map rmirror ts))))
= Node x (rev (rev (map rmirror (map rmirror ts))))
= Node x (map (rmirror ∘ rmirror) ts)                      by IH?
= Node x (map id ts) = Node x ts
```

# Conditional term rewriting

## Congruence rule for map

$$\texttt{?xs = ?ys} \Longrightarrow$$
$$(\bigwedge \texttt{y. y} \in \texttt{set ?ys} \Longrightarrow \texttt{?f y = ?g y}) \Longrightarrow$$
$$\texttt{map ?f ?xs = map ?g ?ys}$$

Assume: `rmirror (rmirror t) = t` for all `t` $\in$ `set ts`

Show: `map (rmirror` $\circ$ `rmirror) ts = map id ts`

$$\frac{}{\texttt{map (rmirror} \circ \texttt{rmirror) ts = ??}}$$

# Conditional term rewriting

## Congruence rule for map

```
?xs = ?ys ⟹
(⋀y. y ∈ set ?ys ⟹ ?f y = ?g y) ⟹
map ?f ?xs = map ?g ?ys
```

Assume: `rmirror (rmirror t) = t` for all `t ∈ set ts`

Show: `map (rmirror ∘ rmirror) ts = map id ts`

$$\dfrac{\texttt{ts = ??} \qquad \dfrac{}{\bigwedge \texttt{t. t} \in \texttt{set ?? } \Longrightarrow \texttt{(rmirror} \circ \texttt{rmirror) t = ?? t}}}{\texttt{map (rmirror} \circ \texttt{rmirror) ts = map ?? ??}}$$

# Conditional term rewriting

## Congruence rule for map

$?xs = ?ys \Longrightarrow$
$(\bigwedge y.\ y \in set\ ?ys \Longrightarrow ?f\ y = ?g\ y) \Longrightarrow$
$map\ ?f\ ?xs = map\ ?g\ ?ys$

Assume: $rmirror\ (rmirror\ t) = t$ for all $t \in set\ ts$

Show: $map\ (rmirror \circ rmirror)\ ts = map\ id\ ts$

$$\dfrac{ts = ts \qquad \dfrac{}{\bigwedge t.\ t \in set\ ts \Longrightarrow (rmirror \circ rmirror)\ t = \ ??\ \ t}}{map\ (rmirror \circ rmirror)\ ts = map\ \ ??\ \ ts}$$

# Conditional term rewriting

## Congruence rule for map

```
?xs = ?ys ⟹
(⋀y. y ∈ set ?ys ⟹ ?f y = ?g y) ⟹
map ?f ?xs = map ?g ?ys
```

Assume: `rmirror (rmirror t) = t` for all `t ∈ set ts`

Show: `map (rmirror ∘ rmirror) ts = map id ts`

$$\frac{\texttt{ts = ts} \qquad \dfrac{}{\bigwedge \texttt{t. t} \in \texttt{set ts} \Longrightarrow \texttt{rmirror (rmirror t)} = \texttt{??}\ \texttt{t}}}{\texttt{map (rmirror} \circ \texttt{rmirror) ts = map}\ \texttt{??}\ \texttt{ts}}$$

# Conditional term rewriting

## Congruence rule for `map`

$\texttt{?xs = ?ys} \Longrightarrow$
$(\bigwedge \texttt{y. y} \in \texttt{set ?ys} \Longrightarrow \texttt{?f y = ?g y}) \Longrightarrow$
$\texttt{map ?f ?xs = map ?g ?ys}$

Assume: $\texttt{rmirror (rmirror t) = t}$ for all $\texttt{t} \in \texttt{set ts}$
Show: $\texttt{map (rmirror} \circ \texttt{rmirror) ts = map id ts}$

$$\dfrac{\texttt{ts = ts} \qquad \dfrac{\bigwedge \texttt{t. t} \in \texttt{set ts} \Longrightarrow \texttt{t} \in \texttt{set ts}}{\bigwedge \texttt{t. t} \in \texttt{set ts} \Longrightarrow \texttt{rmirror (rmirror t)} = (\lambda \texttt{x. x})\ \texttt{t}}}{\texttt{map (rmirror} \circ \texttt{rmirror) ts = map} \ (\lambda \texttt{x. x})\ \texttt{ts}}$$

# Now it's your turn

1. Download `Programming.thy` from the tutorial webpage and open it in Isabelle/jEdit.
2. Define `concat :: 'a list list ⇒ 'a list`
   Find out how `concat` behaves w.r.t. `@` and `rev` and prove it.
3. Define pre-order and post-order traversals for rose trees.
   Prove that `preorder (rmirror t) = rev (postorder t)`.

## Preliminaries

## Programming

## Coprogramming

- Codatatypes
- Primitive Corecursion
- Coinduction

## Advanced Coprogramming

# Types with infinite values

| **type** | **finite values** | **infinite values** |
|---|---|---|
| nat | 0, 1, 2, 3, ... | |
| enat | 0, 1, 2, 3, ... | $\infty$ |

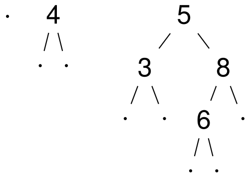
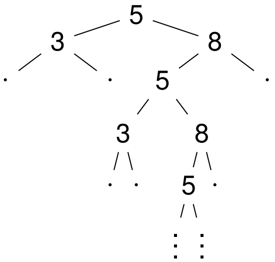
# Types with infinite values

| type | finite values | infinite values |
|---|---|---|
| | $0, S(0), S(S(0)), S(S(S(0))), \ldots$ | |
| nat | $0, 1, 2, 3, \ldots$ | |
| enat | $0, 1, 2, 3, \ldots$ | $\infty$ |
| | $0, S(0), S(S(0)), S(S(S(0))), \ldots$ | $S(S(S(S(S(\ldots)))))$ |

# Types with infinite values

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

# Types with infinite values

| type | finite values | infinite values |
|---|---|---|
| | $0, S(0), S(S(0)), S(S(S(0))), \ldots$ | |
| nat | $0, 1, 2, 3, \ldots$ | |
| enat | $0, 1, 2, 3, \ldots$ | $\infty$ |
| | $0, S(0), S(S(0)), S(S(S(0))), \ldots$ | $S(S(S(S(S(\ldots)))))$ |
| list | $[], [0], [0,0], [0,1,2,3,4], \ldots$ | |
| stream | | $[0,0,0,\ldots], [1,2,3,\ldots], [0,1,0,1,\ldots], \ldots$ |
| llist | $[], [0], [0,0], [0,1,2,3,4], \ldots$ | $[0,0,0,\ldots], [1,2,3,\ldots], [0,1,0,1,\ldots], \ldots$ |
| tree |  |  |

# Codatatypes in Isabelle/HOL

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

# Codatatypes in Isabelle/HOL

discriminator

selector

```
datatype   nat  =           0    |  Suc          nat
codatatype enat = is_zero: Zero | eSuc (epred: enat)
  where "epred Zero = Zero"
```

defaults

# Codatatypes in Isabelle/HOL

discriminator

selector

```
datatype   nat  =            0    |  Suc          nat
codatatype enat = is_zero: Zero | eSuc (epred: enat)
  where "epred Zero = Zero"
```

defaults

```
datatype   'a list   =  Nil |  Cons 'a "'a list"
codatatype 'a stream =        SCons 'a "'a stream"
codatatype 'a llist  = LNil | LCons 'a "'a llist"
```

# Codatatypes in Isabelle/HOL

discriminator

selector

```
datatype   nat  =            0    |  Suc           nat
codatatype enat = is_zero: Zero | eSuc (epred: enat)
  where "epred Zero = Zero"
```

defaults

```
datatype   'a list   =  Nil |  Cons 'a "'a list"
codatatype 'a stream =         SCons 'a "'a stream"
codatatype 'a llist  = LNil | LCons 'a "'a llist"


codatatype 'a tree =
  is_Leaf: Leaf
       | Node (left: 'a tree) (val: 'a) (right: 'a tree)
  where "left Leaf = Leaf"   |   "right Leaf = Leaf"
```

# No recursion on codatatypes

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

Suppose we could do recursion on codatatypes . . .

```
primrec to_nat :: enat ⇒ nat where
  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 . . .

```
primrec to_nat :: enat ⇒ nat where
  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 ∞)
```

$$n \quad = \quad 1 + n$$

False

# Building infinite values by primitive corecursion

**primitive recursion**

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

**primitive corecursion**

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

```
codatatype enat = Zero | eSuc enat

primcorec infty :: enat ("∞") where ∞ = eSuc ∞
```

# Building infinite values by primitive corecursion

**primitive recursion**

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

**primitive corecursion**

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

```
codatatype enat = Zero | eSuc enat

primcorec infty :: enat ("∞") where ∞ = eSuc ∞
```

Derive destructor characterisation:

```
lemma infty.sel:
  "is_zero ∞ = False"
  "epred   ∞ = ∞"
```

# Computing with infinite values

Computing with codatatypes is pattern matching on results:
We can inspect arbitrary finite amounts of output in finitely many steps.

## Addition on enat

```
primcorec eplus :: "enat ⇒ enat ⇒ enat" (infixl "⊕" 65)
where "m ⊕ n = (if is_zero m then n else eSuc (epred m ⊕ n))"
```

# Computing with infinite values

Computing with codatatypes is pattern matching on results:
We can inspect arbitrary finite amounts of output in finitely many steps.

## Addition on enat

```
primcorec eplus :: "enat ⇒ enat ⇒ enat" (infixl "⊕" 65)
where "m ⊕ n = (if is_zero m then n else eSuc (epred m ⊕ n))"
```

corecursive call

# Computing with infinite values

Computing with codatatypes is pattern matching on results:
We can inspect arbitrary finite amounts of output in finitely many steps.

## Addition on enat

```
primcorec eplus :: "enat ⇒ enat ⇒ enat" (infixl "⊕" 65)
where "m ⊕ n = (if is_zero m then n else eSuc (epred m ⊕ n))"
```

guard

corecursive call

corecursive argument

# Computing with infinite values

Computing with codatatypes is pattern matching on results:
We can inspect arbitrary finite amounts of output in finitely many steps.

## Addition on enat

```
primcorec eplus :: "enat ⇒ enat ⇒ enat" (infixl "⊕" 65)
where "m ⊕ n = (if is_zero m then n else eSuc (epred m ⊕ n))"
```

corecursion stops

guard

corecursive argument

corecursive call

# Computing with infinite values

Computing with codatatypes is pattern matching on results:
We can inspect arbitrary finite amounts of output in finitely many steps.

## Addition on enat

```
primcorec eplus :: "enat ⇒ enat ⇒ enat" (infixl "⊕" 65)
where "m ⊕ n = (if is_zero m then n else eSuc (epred m ⊕ n))"
```

corecursion stops

guard

corecursive call

corecursive argument

```
lemma infty.sel:
  "is_zero ∞ = False"
  "epred   ∞ = ∞"
```

## Evaluate $\infty \oplus 3$

# Computing with infinite values

Computing with codatatypes is pattern matching on results:
We can inspect arbitrary finite amounts of output in finitely many steps.

## Addition on enat

```
primcorec eplus :: "enat ⇒ enat ⇒ enat" (infixl "⊕" 65)
where "m ⊕ n = (if is_zero m then n else eSuc (epred m ⊕ n))"
```

corecursion stops

guard

corecursive argument

corecursive call

```
lemma infty.sel:
  "is_zero ∞ = False"
  "epred   ∞ = ∞"
```

## Evaluate $\infty \oplus 3$ by observing!

```
is_zero (      ∞ ⊕ 3) =
```

# Computing with infinite values

Computing with codatatypes is pattern matching on results:
We can inspect arbitrary finite amounts of output in finitely many steps.

## Addition on enat

```
primcorec eplus :: "enat ⇒ enat ⇒ enat" (infixl "⊕" 65)
where "m ⊕ n = (if is_zero m then n else eSuc (epred m ⊕ n))"
```

corecursion stops

guard

corecursive argument

corecursive call

```
lemma infty.sel:
  "is_zero ∞ = False"
  "epred   ∞ = ∞"
```

## Evaluate $\infty \oplus 3$ by observing!

```
is_zero (      ∞ ⊕ 3) = False
```

# Computing with infinite values

Computing with codatatypes is pattern matching on results:
We can inspect arbitrary finite amounts of output in finitely many steps.

## Addition on enat

```
primcorec eplus :: "enat ⇒ enat ⇒ enat" (infixl "⊕" 65)
where "m ⊕ n = (if is_zero m then n else eSuc (epred m ⊕ n))"
```

corecursion stops

guard

corecursive call

corecursive argument

```
lemma infty.sel:
  "is_zero ∞ = False"
  "epred   ∞ = ∞"
```

## Evaluate ∞ ⊕ 3 by observing!

```
is_zero (      ∞ ⊕ 3) = False
epred   (      ∞ ⊕ 3) =
```

# Computing with infinite values

Computing with codatatypes is pattern matching on results:
We can inspect arbitrary finite amounts of output in finitely many steps.

## Addition on enat

```
primcorec eplus :: "enat ⇒ enat ⇒ enat" (infixl "⊕" 65)
where "m ⊕ n = (if is_zero m then n else eSuc (epred m ⊕ n))"
```

corecursion stops

guard

corecursive argument

corecursive call

```
lemma infty.sel:
  "is_zero ∞ = False"
  "epred   ∞ = ∞"
```

## Evaluate ∞ ⊕ 3 by observing!

```
is_zero (      ∞ ⊕ 3) = False
epred   (      ∞ ⊕ 3) = epred ∞ ⊕ 3
```

# Computing with infinite values

Computing with codatatypes is pattern matching on results:
We can inspect arbitrary finite amounts of output in finitely many steps.

## Addition on enat

```
primcorec eplus :: "enat ⇒ enat ⇒ enat" (infixl "⊕" 65)
where "m ⊕ n = (if is_zero m then n else eSuc (epred m ⊕ n))"
```

corecursion stops

guard

corecursive call

corecursive argument

```
lemma infty.sel:
  "is_zero ∞ = False"
  "epred   ∞ = ∞"
```

## Evaluate ∞ ⊕ 3 by observing!

```
is_zero (      ∞ ⊕ 3) = False
epred   (      ∞ ⊕ 3) = epred ∞ ⊕ 3
is_zero (epred ∞ ⊕ 3) =
```

# Computing with infinite values

Computing with codatatypes is pattern matching on results:
We can inspect arbitrary finite amounts of output in finitely many steps.

## Addition on enat

```
primcorec eplus :: "enat ⇒ enat ⇒ enat" (infixl "⊕" 65)
where "m ⊕ n = (if is_zero m then n else eSuc (epred m ⊕ n))"
```

corecursion stops

guard

corecursive argument

corecursive call

```
lemma infty.sel:
  "is_zero ∞ = False"
  "epred   ∞ = ∞"
```

## Evaluate ∞ ⊕ 3 by observing!

```
is_zero (      ∞ ⊕ 3) = False
epred   (      ∞ ⊕ 3) = epred ∞ ⊕ 3
is_zero (epred ∞ ⊕ 3) = is_zero (∞ ⊕ 3) = False
epred   (epred ∞ ⊕ 3) = ...
```

# Computing with infinite values

Computing with codatatypes is pattern matching on results:
We can inspect arbitrary finite amounts of output in finitely many steps.

## Addition on enat

```
primcorec eplus :: "enat ⇒ enat ⇒ enat" (infixl "⊕" 65)
where "m ⊕ n = (if is_zero m then n else eSuc (epred m ⊕ n))"
```

corecursion stops

guard

corecursive argument

corecursive call

```
lemma infty.sel:
  "is_zero ∞ = False"
  "epred   ∞ = ∞"
```

**Conjecture: $\infty \oplus 3 = \infty$**

## Evaluate $\infty \oplus 3$ by observing!

```
is_zero (      ∞ ⊕ 3) = False
epred   (      ∞ ⊕ 3) = epred ∞ ⊕ 3
is_zero (epred ∞ ⊕ 3) = is_zero (∞ ⊕ 3) = False
epred   (epred ∞ ⊕ 3) = ...
```

# When are two extended naturals equals?

## If we can make the same observations!

$2 \quad =? \quad 2 \oplus 0$

# When are two extended naturals equals?

## If we can make the same observations!

| | | |
|---|---|---|
| $2$ | =? | $2 \oplus 0$ |
| $\text{is-zero } 2 = \text{False}$ | = | $\text{is-zero } (1 \oplus 0) = \text{False}$ |
| $\text{epred } 2 = 1$ | =? | $\text{epred } (2 \oplus 0) = \text{epred } 2 \oplus 0 = 1 \oplus 0$ |

# When are two extended naturals equals?

**If we can make the same observations!**

$$
\begin{array}{lll}
2 & =? & 2 \oplus 0 \\
\\
\text{is-zero } 2 = \text{False} & = & \text{is-zero } (1 \oplus 0) = \text{False} \\
\text{epred } 2 = 1 & =? & \text{epred } (2 \oplus 0) = \text{epred } 2 \oplus 0 = 1 \oplus 0 \\
\\
\text{is-zero } 1 = \text{False} & = & \text{is-zero } (1 \oplus 0) = \text{False} \\
\text{epred } 1 = 0 & =? & \text{epred } (1 \oplus 0) = \text{epred } 1 \oplus 0 = 0 \oplus 0
\end{array}
$$

# When are two extended naturals equals?

**If we can make the same observations!**

| | | |
|---|---|---|
| $2$ | $=?$ | $2 \oplus 0$ |
| $\text{is-zero } 2 = \text{False}$ | $=$ | $\text{is-zero } (1 \oplus 0) = \text{False}$ |
| $\text{epred } 2 = 1$ | $=?$ | $\text{epred } (2 \oplus 0) = \text{epred } 2 \oplus 0 = 1 \oplus 0$ |
| $\text{is-zero } 1 = \text{False}$ | $=$ | $\text{is-zero } (1 \oplus 0) = \text{False}$ |
| $\text{epred } 1 = 0$ | $=?$ | $\text{epred } (1 \oplus 0) = \text{epred } 1 \oplus 0 = 0 \oplus 0$ |
| $\text{is-zero } 0 = \text{True}$ | $=$ | $\text{is-zero } (0 \oplus 0) = \text{True}$ |

# When are two extended naturals equals?

## If we can make the same observations!

| | | |
|---|---|---|
| $2$ | =? | $2 \oplus 0$ |
| is-zero $2 = \text{False}$ | = | is-zero $(1 \oplus 0) = \text{False}$ |
| epred $2 = 1$ | =? | epred $(2 \oplus 0) =$ epred $2 \oplus 0 = 1 \oplus 0$ |
| is-zero $1 = \text{False}$ | = | is-zero $(1 \oplus 0) = \text{False}$ |
| epred $1 = 0$ | =? | epred $(1 \oplus 0) =$ epred $1 \oplus 0 = 0 \oplus 0$ |
| is-zero $0 = \text{True}$ | = | is-zero $(0 \oplus 0) = \text{True}$ |

## Coinduction rule for equality

$$\frac{R\ n\ m \qquad \begin{array}{l} \forall n\ m.\ R\ n\ m \longrightarrow \text{is-zero}\ n = \text{is-zero}\ m \land \\ \qquad\qquad\qquad (\neg \text{is-zero}\ m \longrightarrow R\ (\text{epred}\ n)\ (\text{epred}\ m)) \end{array}}{n = m}$$

# When are two extended naturals equals?

## If we can make the same observations!

| | | |
|---|---|---|
| $2$ | R | $2 \oplus 0$ |
| is-zero $2 =$ False | = | is-zero $(1 \oplus 0) =$ False |
| epred $2 = 1$ | R | epred $(2 \oplus 0) =$ epred $2 \oplus 0 = 1 \oplus 0$ |
| is-zero $1 =$ False | = | is-zero $(1 \oplus 0) =$ False |
| epred $1 = 0$ | R | epred $(1 \oplus 0) =$ epred $1 \oplus 0 = 0 \oplus 0$ |
| is-zero $0 =$ True | = | is-zero $(0 \oplus 0) =$ True |

## Coinduction rule for equality

$$\frac{R\ n\ m \qquad \begin{array}{l} \forall n\ m.\ R\ n\ m \longrightarrow \text{is-zero}\ n = \text{is-zero}\ m\ \wedge \\ \qquad\qquad\qquad (\neg \text{is-zero}\ m \longrightarrow R\ (\text{epred}\ n)\ (\text{epred}\ m)) \end{array}}{n = m}$$

# Prove that $\infty \oplus x = \infty$

## Coinduction rule for equality

$$\frac{R\ n\ m \qquad \begin{array}{l} \forall n\ m.\ R\ n\ m \longrightarrow \text{is-zero}\ n = \text{is-zero}\ m \wedge \\ \qquad (\neg \text{is-zero}\ m \longrightarrow R\ (\text{epred}\ n)\ (\text{epred}\ m)) \end{array}}{n = m}$$

1. Define $R\ n\ m \longleftrightarrow n = \infty \oplus x \wedge m = \infty$
2. Show that $R\ (\infty \oplus x)\ \infty$.
3. Show bisimulation property of $R$:
   - Assume $R\ n\ m$ for arbitrary $n$, $m$.
     So $n = \infty \oplus x$ and $m = \infty$.
   - is-zero $n =$ is-zero $(\infty \oplus x) =$ False $=$ is-zero $\infty$
   - Show $R$ (epred $n$) (epred $m$):
     - epred $n =$ epred $(\infty \oplus x) =$ epred $\infty \oplus x$
     - epred $m =$ epred $\infty = \infty$

# Isabelle demo

# From lazy lists to trees and back

```
codatatype 'a llist =
  lnull: LNil
       | LCons (lhd: 'a)
               (ltl: 'a llist)
```

$\xrightarrow{\texttt{tree\_of}}$

$\xleftarrow{\texttt{llist\_of}}$

```
codatatype 'a tree =
  is_Leaf: Leaf
         | Node (left: 'a tree)
                (val: 'a)
                (right: 'a tree)
```

$[0, 1, 2, \ldots] \xrightarrow{\texttt{tree\_of}}$

# From lazy lists to trees and back

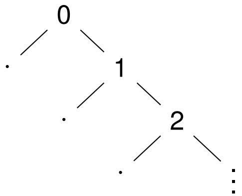
```
codatatype 'a llist =
  lnull: LNil
       | LCons (lhd: 'a)
               (ltl: 'a llist)
```

$\xrightarrow{\texttt{tree\_of}}$

$\xleftarrow{\texttt{llist\_of}}$

```
codatatype 'a tree =
  is_Leaf: Leaf
         | Node (left: 'a tree)
                (val: 'a)
                (right: 'a tree)
```

$[0, 1, 2, \ldots] \xrightarrow{\texttt{tree\_of}}$



# From lazy lists to trees and back

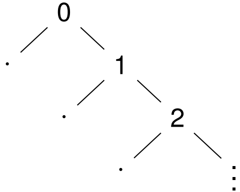
```
codatatype 'a llist =
  lnull: LNil
       | LCons (lhd: 'a)
               (ltl: 'a llist)
```

tree_of $\longrightarrow$

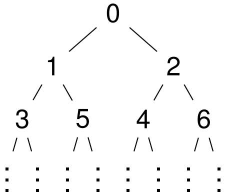
llist_of $\longleftarrow$

```
codatatype 'a tree =
  is_Leaf: Leaf
         | Node (left: 'a tree)
                (val: 'a)
                (right: 'a tree)
```

$[0, 1, 2, ...]$ $\xrightarrow{\texttt{tree\_of}}$

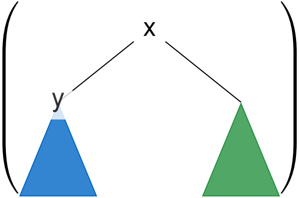



$[0, 1, 2, 3, 4, 5, 6, \ldots]$ $\xleftarrow{\texttt{llist\_of}}$



# Tree chopping

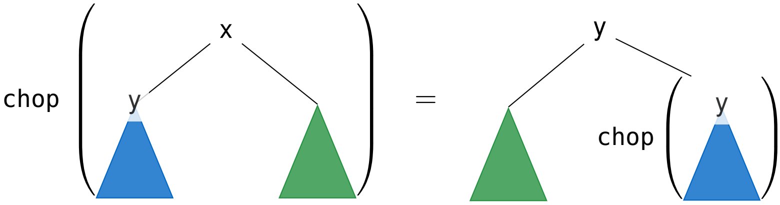
Remove the root of a tree:

chop 

=

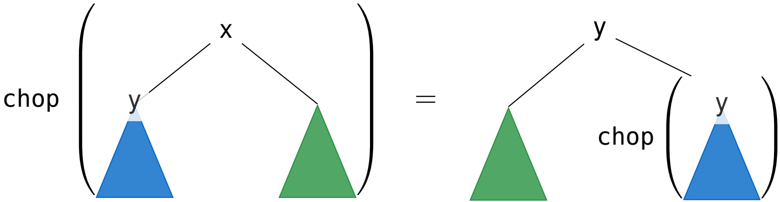
# Tree chopping

Remove the root of a tree:



# Tree chopping

Remove the root of a tree:




What if there is a `Leaf`?

Preliminaries 

Programming 

Coprogramming 

**Advanced Coprogramming** 

- Corecursion Up To Friends
- Coinduction Up To Friends
- Mixed Recursion-Corecursion

# Am I productive?

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

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

primitive corecusion

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

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

`stl` evil

```
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 ⊕ t = (shd s + shd t) ## (stl s ⊕ stl t)
```

```
s ⊕ t = (shd s + shd t) ## (stl s ⊕ stl t)
```

primitive corecusion

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

```
s ⊗ t = (shd s * shd t) ## (stl s ⊗ t ⊕ s ⊗ 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

$$s = 0 \;\#\#\; ((1 \;\#\#\; s) \oplus s)$$

```
s = 0 ## ((1 ## s) ⊕ s)
```

corecursion up-to constructors and $\oplus$

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

```
s = (0 ## 1 ## s) ⊕ (0 ## s)
```

corecursion up to constructors and $\oplus$

$\oplus$ comes before the guard

$$s = (1 \ \#\# \ s) \otimes (1 \ \#\# \ s)$$

$$s = (1 \ \#\# \ s) \otimes (1 \ \#\# \ s)$$

corecursion up to constructors and $\otimes$

$\otimes$ comes before the guard

```
pow2 t = (2 ^ shd t) ## (stl s ⊗ pow2 t)
```

```
pow2 t = (2 ^ shd t) ## (stl s ⊗ pow2 t)
```

corecursion up to $\otimes$

```
s = pow2 (0 ## s)
```

```
s = pow2 (0 ## s)
```

corecursion up to constructors and `pow2`

`pow2` 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 ⊕ 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 s (n - 1) ⊕ (0 ## s (n + 1))
       else 1 ## s 1
```

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

mixed recursion/corecursion up to ⊕

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

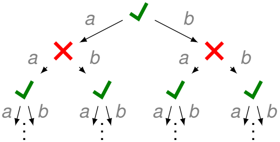
```
s n =  if n > 0
       then stl (s (n - 1)) ⊕ (0 ## s (n + 1))
       else 1 ## s 1
```

stl really evil

# Isabelle demo

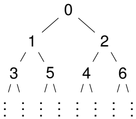
# Exercises

## Languages as Infinite Tries ★




## Growing a Tree ★★

$[0, 1, 2, 3, 4, 5, 6, \ldots] \xleftarrow{\texttt{llist\_of}}$




## Factorial via Shuffle ★★★

Prove that `s = 1 ## (s ⊗ s)` defines the stream of factorials.

A glimpse

BEHIND THE SCENES

with