Verification of Functional Programs

Andrés Sicard-Ramírez

EAFIT University

Semester 2014-1
Administrative Information

Course web page
http://www1.eafit.edu.co/asr/courses/fpv-CB0683/

Evaluation

Homework    30%
Presentation 30%
Final project 40%
Motivation

U$22.2 to U$59.5 billion!*

Motivational Example

“Every functional programmer worth his salt knows how to reverse a list, debug the code, and prove that list reversal is its own inverse.”*

“Every functional programmer \textit{worth his salt} knows how to reverse a list, debug the code, and prove that list reversal is its own inverse.”* 

Let’s go (Haskell code) …

\[
(++) \colon \text{[a]} \rightarrow \text{[a]} \rightarrow \text{[a]} \\
\text{[]} ++ \text{ys} = \text{ys} \\
(x : \text{xs}) ++ \text{ys} = x : (\text{xs} ++ \text{ys}) \\
\]

\[
\text{rev} \colon \text{[a]} \rightarrow \text{[a]} \\
\text{rev} \text{[]} = \text{[]} \\
\text{rev} (x : \text{xs}) = \text{rev} \text{xs} ++ [x] \\
\]

To prove that the \text{rev} function is an involution.

Motivational Example

Example
Proving \( \text{rev } (\text{rev } x) = x \).

Case \([]\).

\[
\text{rev } (\text{rev } []) = \text{rev } [] \quad (\text{rev.1}) \\
= [] \quad (\text{rev.1})
\]
Motivational Example

Example

Proving \( \text{rev} (\text{rev} \; xs) = xs \).

Case \([\,]\).

\[
\text{rev (rev \; [\,])} = \text{rev} \; [\,] \quad \text{(rev.1)}
\]
\[
= [\,] \quad \text{(rev.1)}
\]

Case \(x:xs\).

\[
\text{rev (rev \; (x : xs))} = \text{rev (rev \; xs \; \text{++} \; [x])} \quad \text{(rev.2)}
\]
\[
= x : \text{rev (rev \; xs)} \quad \text{(auxiliary thm.)}
\]
\[
= x : xs \quad \text{(IH)}
\]

Auxiliary theorem: \( \text{rev} \; (ys \; \text{++} \; [x]) = x : \text{rev} \; ys \).
Motivational Example

Observation

The auxiliary theorem

\[ \text{rev} (\text{ys} ++ [x]) = x : \text{rev} \ \text{ys} \]

is a generalisation of the required result

\[ \text{rev} (\text{rev} \ \text{x} \ ++ [x]) = x : \text{rev} (\text{rev} \ \text{x}). \]

“A standard method of generalisation is to look for a sub-expression that appears on both sides of the equation and replace it by a variable.”*  

Observations from the Motivational Example

Inductive data types $\Rightarrow$ **Structural induction** for reasoning about them.
Observations from the Motivational Example

- Inductive data types $\Rightarrow$ **Structural induction** for reasoning about them.

- **Equational reasoning** (process of replacing like for like using the substitutivity property and the equivalence properties of the equality) based on the **referential transparency**.
Observations from the Motivational Example

- Inductive data types ⇒ Structural induction for reasoning about them.

- Equational reasoning (process of replacing like for like using the substitutivity property and the equivalence properties of the equality) based on the referential transparency.

- Generalisation of auxiliary theorem (including the inductive hypothesis) ⇒ Proofs by induction are difficulty to automatise.
Questions from the Motivational Example

- What about \( \bot \)?

\[
\text{rev (rev } \bot) \equiv \bot
\]

---

Questions from the Motivational Example

- What about $\bot$?

  $\text{rev} \ (\text{rev} \ \bot) \cong \bot$

- Extend structural induction for handling $\bot$.

---


Questions from the Motivational Example

- What about ⊥?
  \[ \text{rev} (\text{rev} \; ⊥) \equiv ⊥ \]

- **Extend** structural induction for handling ⊥.

- Choose a programming logic to behaviours of programs on total and finite elements of data structures.

---


Questions from the Motivational Example

- What about $\bot$?

$$\text{rev} \ (\text{rev} \ \bot) \cong \bot$$

- Extend structural induction for handling $\bot$.

- Choose a programming logic to behaviours of programs on total and finite elements of data structures. $^{*\dagger}$

- “Morally” correct reasoning. $^{\dagger}$

---


Questions from the Motivational Example

- What about if `xs` is an infinite list?

\[ \text{rev (rev } xs) = xs \]

---


‡Ana Bove, Peter Dybjer and Andrés Sicard-Ramírez (2012). Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs.
Questions from the Motivational Example

- What about if xs is an infinite list?
  \[ \text{rev (rev } xs) \equiv xs \]

- Co-inductive data types $\Rightarrow$ Co-induction for reasoning about them.

---

‡Ana Bove, Peter Dybjer and Andrés Sicard-Ramírez (2012). Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs.
Questions from the Motivational Example

- What about if \( xs \) is an infinite list?

\[
\text{rev (rev xs)} \equiv xs
\]

- Co-inductive data types \( \Rightarrow \text{Co-induction} \) for reasoning about them.*

- Choose a programming logic to behaviours of programs on total (finite or potentially unbounded) elements of data structures.†‡

---

‡ Ana Bove, Peter Dybjer and Andrés Sicard-Ramírez (2012). Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs.
Questions from the Motivational Example

- The `rev` function is $O(n^2)$. Why are we reasoning about it?

  GHCi> rev [1..10^7]
  *** Exception: stack overflow
The rev function is $O(n^2)$. Why are we reasoning about it?

```
GHCi> rev [1..10^7]
*** Exception: stack overflow
```

The reverse function in the Data.List library (GHC 7.8.2) is $O(n)$:

```haskell
reverse l = rev l []
where
  rev [] a = a
  rev (x:xs) a = rev xs (x:a)
```
In relation to the formal verification of find or gcd algorithms versus the verification of real programs:

“They are differences in kind. Babysitting for a sleeping child for one hour does not scale up to raising a family of ten—the problems are essentially, fundamentally different.”

### Verification of Functional Programs: Research Areas

<table>
<thead>
<tr>
<th>Area</th>
<th>Research focuses on</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics definitions</td>
<td>Defining new concepts</td>
</tr>
<tr>
<td>Transformation rules</td>
<td>Programming transformations</td>
</tr>
<tr>
<td>Functional properties verification</td>
<td>The input and output correspondence of programs</td>
</tr>
<tr>
<td>Non-functional properties verification</td>
<td>Properties such as memory consumption or parallel performance</td>
</tr>
</tbody>
</table>

---


1968  “In 1968, a NATO Conference on Software Engineering was held in Garmisch, Germany, ...For the first time, a consensus emerged that there really was a software crisis, that programming was not very well understood.”*


Preliminary Concepts
What is a Type?

- A type is a set of values (and operations on them).

*Bertrand Russell (1908). Mathematical Logic as based on the Theory of Types.*
What is a Type?

- A type is a set of values (and operations on them).
- Types as ranges of significance of propositional functions.*

In modern terminology, types are domains of predicates.

Example

\[
P(0) \quad \forall x. P(x) \rightarrow P(x') \\
\frac{}{\forall x. P(x)}
\]

The above rule make sense only when \( P \) is a predicate over natural numbers (i.e. \( P : \mathbb{N} \rightarrow \{T, F\} \)).

*Bertrand Russell (1908). Mathematical Logic as based on the Theory of Types.
What is a Type?

“A type is an approximation of a dynamic behaviour that can be derived from the form of an expression.”*

________________________
What is a Type?

- “A type is an approximation of a dynamic behaviour that can be derived from the form of an expression.”*

- The propositions-as-types principle (Curry-Howard correspondence)

What is a Type?

- “A type is an approximation of a dynamic behaviour that can be derived from the form of an expression.”*

- The propositions-as-types principle (Curry-Howard correspondence)

- Homotopy Type Theory (HTT)

  Propositions are types, but not all types are propositions (e.g. higher-order inductive types)

What is a Type?

Example (some Haskell’s types)

- Type variables: \( a, b \)
- Type constants: \( \text{Int}, \text{Integer}, \text{Char} \)
- Function types: \( \text{Int} \to \text{Bool}, (\text{Char} \to \text{Int}) \to \text{Integer} \)
- Product types: \( (\text{Int}, \text{Char}), (a, b) \)
- Disjoint union types:

\[
\text{data Sum } a \ b = \text{Inl } a \mid \text{Inr } b
\]
Over-sized slogan:

"Well-type programs cannot "go wrong""*

---

†Benjamin C. Pierce (2002). Types and Programming Languages, p. 1.
Over-sized slogan:

“Well-type programs cannot “go wrong””*

“A type system is a *tractable* syntactic method for proving the absence of *certain* program behaviors by classifying phrases according to the kinds of values they compute.”†

---

†Benjamin C. Pierce (2002). Types and Programming Languages, p. 1.
Referential Transparency

“We use [referential transparency] to refer to the fact of mathematics which says: The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value.”*

Referential Transparency

“We use [referential transparency] to refer to the fact of mathematics which says: The only thing that matters about an expression is its value, and any subexpression can be replaced by any other equal in value.”*

“A language that supports the concept that “equals can be substituted for equals” in an expression without changing the value of the expression is said to be referentially transparent.”†

Example
The following C program prints hello, world twice.

```c
#include <stdio.h>

int main (void)
{
    printf ("hello, world");
    printf ("hello, world");

    return 0;
}
```
Example
The following C program prints hello, world once.

```c
#include <stdio.h>

int main (void)
{
    int x;
    x = printf ("hello, world");
    x; x;

    return 0;
}
```
Example

The following Haskell program prints hello, world twice.

```haskell
main :: IO ()
main = putStr "hello, world" >> putStr "hello, world"
```
In Haskell, given

```haskell
let x = exp
in  ... x ... x ...
```

the meaning of ... x ... x ... is the same as ... exp ... exp ...
Referential Transparency

In Haskell, given

```haskell
let x = exp
in ... x ... x ...
```

the meaning of `... x ... x ...` is the same as `... exp ... exp ...`

Example
The following Haskell program prints hello, world twice.

```haskell
main :: IO ()
main = let x :: IO ()
      x = putStrLn "hello, world"
      in x >>= x
```
Example

The following Haskell program prints hello, world twice.

```haskell
main :: IO ()
main = x >> x
  where x :: IO ()
        x = putStrLn "hello, world"
```
Pure Functions

Side effects

“A side effect introduces a dependency between the global state of the system and the behaviour of a function … Side effects are essentially invisible inputs to, or outputs from, functions.”

* Bryan O'Sullivan, John Goerzen and Don Stewart (2008). Real World Haskell, p. 27.
Side effects

“A side effect introduces a dependency between the global state of the system and the behaviour of a function … Side effects are essentially invisible inputs to, or outputs from, functions.” *

Pure functions

“Take all their input as explicit arguments, and produce all their output as explicit results.”†

Are the following GHC 7.8.2 functions, pure functions?

```haskell
maxBound :: Int -- Prelude
os :: String -- System.Info
```
Pure Functions

Are the following GHC 7.8.2 functions, pure functions?

maxBound :: Int -- Prelude
os :: String -- System.Info

“One perspective is that Haskell is not just one language (plus Prelude), but a family of languages, parametrized by a collection of implementation-dependent parameters. Each such language is RT, even if the collection as a whole might not be. Some people are satisfied with situation and others are not.”

Functions are First-Class Citizens*

- They can be passed as arguments and they can be returned as results (higher-order functions)
- They can be assigned to variables
- They can be stored in data structures

Working with functions how handle undefined values yielded by partial functions or non-terminating functions?

Example

```
head :: [a] → a
head (x : _) = x

head [] = ?
```
Working with **functions** how handle undefined values yielded by **partial** functions or **non-terminating** functions?

**Example**

```haskell
head :: [a] → a
head (x : _) = x

head [] = ?
```

**Example**

```haskell
fst :: (a, b) → a
fst (x, _) = x

ones :: [Int]
ones = 1 : ones

fst (ones, 10) = ?
```
The $\bot$ symbol represents the undefined value. ($\bot$ is represented in Haskell by the `undefined` keyword)

Example (first version)

```haskell
head [] = undefined
fst (ones, 10) = undefined
```

Remark

The $\bot$ value is polymorphic in Haskell.

Remark

The Haskell types are lifted types.

See “Hussling Haskell types into Hasse diagrams” from Edward Z. Yang’s blog on December 6, 2010.
The $\bot$ symbol represents the undefined value. ($\bot$ is represented in Haskell by the `undefined` keyword)

Example (first version)

```haskell
head [] = undefined
fst (ones, 10) = undefined
```

Remark

The $\bot$ value is polymorphic in Haskell.

Remark

The Haskell types are lifted types*

*See “Hussling Haskell types into Hasse diagrams” from Edward Z. Yang's blog on December 6, 2010.
Example (second version)

\[
\text{head } [] = \bot_a \\
\text{fst (ones, 10)} = \bot_{\text{Int}}
\]

Therefore, head [] \neq \text{fst (ones, 10)}. 
Example

\[
\text{foo} :: \text{Int} \to \text{Int} \\
\text{foo} \ 0 = 0
\]

\[
\text{bar} :: \text{Int} \to \text{Int} \\
\text{bar} \ n = \text{bar} \ (n + 1)
\]

\[
\text{foobar} :: \text{Int} \to \text{Int} \\
\text{foobar} \ n = \text{if} \ \text{foo} \ n == 0 \ \text{then} \ 1 \ \text{else} \ 2
\]
Example

```haskell
foo :: Int → Int
foo 0 = 0

bar :: Int → Int
bar n = bar (n + 1)

foobar :: Int → Int
foobar n = if foo n == 0 then 1 else 2
```

Can we replace `foo` by `bar` in `foobar`?
Example

foo :: Int → Int
foo 0 = 0

bar :: Int → Int
bar n = bar (n + 1)

foobar :: Int → Int
foobar n = if foo n == 0 then 1 else 2

Can we replace foo by bar in foobar? Only for $n \neq 0$. 
Lazy Evaluation


Strict and Non-Strict Functions

Definition
Let $f$ be a unary function. If $f \bot = \bot$ then $f$ is a strict function, otherwise it is a non-strict function. The definition generalise to $n$-ary functions.

Example
The three function is non-strict.

```haskell
three :: a -> Int
three _ = 3
three undefined = 3
three (head []) = 3
three (fst (ones, 10)) = 3
three (putStr "hello, world") = 3
```
Strict and Non-Strict Functions

Example

\[
\text{three :: } a \rightarrow \textbf{Int} \\
\text{three } _ = 3
\]

Non-strict reasoning...

\[
\forall x \in \text{Int.} \forall y. \ x + \text{three } y = x + 3.
\]
Strict and Non-Strict Functions

Example
(Why Haskell hasn’t a predefined recursive data type for natural numbers?)

```
data Nat = Zero | Succ Nat
Zero :: Nat
Succ :: Nat → Nat
```
Example

(Why Haskell hasn’t a predefined recursive data type for natural numbers?)

```haskell
data Nat = Zero | Succ Nat

Zero :: Nat
Succ :: Nat → Nat
```

Is Succ a non-strict function?
Strict and Non-Strict Functions

Example

(Why Haskell hasn’t a predefined recursive data type for natural numbers?)

```haskell
data Nat = Zero | Succ Nat
Zero :: Nat
Succ :: Nat → Nat
```

Is Succ a non-strict function?

We can define

```haskell
inf :: Nat
inf = Succ inf
```
Example (cont.)

Example (cont.)

Nat represents the lazy natural numbers, that is, Succ ⊥ ≠ ⊥.*

\[
\begin{align*}
0 &= ⊥, \\
n + 1 &= \text{Succ } n, \\
\inf &= \bigsqcup_{n ∈ \omega} n
\end{align*}
\]

Partially Ordered Sets

Definition (Partially ordered set)

A partially ordered set (poset) \((D, \sqsubseteq)\) is a set \(D\) on which the binary relation \(\sqsubseteq\) satisfies the following properties:

\[
\forall x. x \sqsubseteq x \quad \text{(reflexive)}
\]

\[
\forall x y z. x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z \quad \text{(transitive)}
\]

\[
\forall x y. x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y \quad \text{(antisymmetry)}
\]

Examples

- \((\mathbb{Z}, \leq)\) is a poset.
- Let \(a, b \in \mathbb{Z}\) with \(a \neq 0\). The divisibility relation is defined by \(a \mid b \overset{\text{def}}{=} \exists c (ac = b)\). Then \((\mathbb{Z}^+, \mid)\) is a poset.
- \((P(A), \subseteq)\) is a poset.
Example

Hasse diagram for the poset \((\{1, 2, 3, 4, 6, 8, 12\}, |)\).
Example

Hasse diagram for the poset \((\{a, b, c\}, \subseteq)\).
Monotone Functions

Definition (Monotone function)

Let \((D, \sqsubseteq)\) and \((D', \sqsubseteq')\) be two posets. A function \(f : D \to D'\) is monotone if

\[
\forall x \; y. \; x \sqsubseteq y \implies f(x) \sqsubseteq' f(y).
\]
Let $D$ be a set, $(D, \sqsubseteq)$ be a poset and $f$ be a function $f : D \to D$.

**Definition (Fixed-point)**

An element $d \in D$ is a fixed-point of $f$ if $f(d) = d$.
Some Concepts of Fixed-Point Theory

Let $D$ be a set, $(D, \sqsubseteq)$ be a poset and $f$ be a function $f : D \rightarrow D$.

**Definition (Fixed-point)**
An element $d \in D$ is a fixed-point of $f$ if $f(d) = d$.

**Definition (Least/Greatest fixed-point)**
The least/greatest fixed-point of $f$ is least/greatest among the fixed-points of $f$. That is, $d \in D$ is the least/greatest fixed-point of $f$ if:
- $f(d) = d$ and $\forall x. f(x) = x \Rightarrow d \sqsubseteq x$
- $\forall x. f(x) = x \Rightarrow x \sqsubseteq d$.
Let $D$ be a set, $(D, \sqsubseteq)$ be a poset and $f$ be a function $f : D \rightarrow D$.

**Definition (Fixed-point)**

An element $d \in D$ is a fixed-point of $f$ if $f(d) = d$.

**Definition (Least/Greatest fixed-point)**

The least/greatest fixed-point of $f$ is least/greatest among the fixed-points of $f$.

That is, $d \in D$ is the least/greatest fixed-point of $f$ if:

- $f(d) = d$ and
- $\forall x. f(x) = x \Rightarrow d \sqsubseteq x$ / $\forall x. f(x) = x \Rightarrow x \sqsubseteq d$. 
Theorem

Let \((D, \sqsubseteq)\) be a poset and \(f : D \rightarrow D\) be monotone. Under certain conditions \(f\) has a least fixed-point* and a greatest fixed-point.†

Some Concepts of Fixed-Point Theory

Theorem

Let \((D, \sqsubseteq)\) be a poset and \(f : D \rightarrow D\) be monotone. Under certain conditions \(f\) has a least fixed-point* and a greatest fixed-point.†

Notation

The least and greatest fixed-points of \(f\) are denoted by \(\mu x.f(x)\) and \(\nu x.f(x)\), respectively.

Motivation: Does λ-calculus have models?

“Historically my first model for the λ-calculus was discovered in 1969 and details were provided in Scott (1972)* (written in 1971).”†

---

* Dana Scott (1972). Continuous Lattices.
Non-standard definitions
pre-domain, domain, complete partial order (cpo), $\omega$-cpo, bottomless $\omega$-cpo, Scott’s domain, ...

Convention
domain $\equiv \omega$-complete partial order
Definition ($\omega$-chain)

Let $(D, \sqsubseteq)$ be a poset. A $\omega$-chain of $D$ is an increasing chain

\[ d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots \]

where $d_i \in D$. 
Definition ($\omega$-complete partial order)

poset $D$ is a $\omega$-complete partial order* ($\omega$-cpo) if

1. There is a least element $\bot \in D$, that is, $\forall x . \bot \sqsubseteq x$. The element $\bot$ is called \textit{bottom}.

2. For every increasing $\omega$-chain $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$, the least upper bound $\bigsqcup_{n \in \omega} d_n \in D$ exists.

\( \omega \)-Complete Partial Orders

**Definition (Lifted set)**

\( A_\perp \) denotes the \( \omega \)-cpo whose elements \( A \cup \{ \perp \} \) are ordered by \( x \sqsubseteq y \), if and only if, \( x = \perp \) or \( x = y \). The \( \omega \)-cpo \( A_\perp \) is called \( A \) lifted.*

Examples

The lifted unit type and the lifted Booleans $B_\bot$ are $\omega$-cpos.

```
data () = ()
data Bool = True | False
```
Example
The lifted natural numbers $\mathbb{N}_\bot$. 

0 1 2 ... n ...
Example

The lazy natural numbers ω-cpo.

```haskell
data Nat = Zero | Succ Nat
```

\[ \begin{align*}
0 = \bot, \\
{n + 1} = \text{Succ } n, \\
\inf = \bigsqcup_{n \in \omega} n
\end{align*} \]

\[ \bigsqcup_{n \in \omega} n = \bot \sqsubseteq \text{Succ } \bot \sqsubseteq \text{Succ } (\text{Succ } \bot) \sqsubseteq \ldots \]

\[ = \inf = \text{Succ } \inf \]
Definition (Admissible property)

Let $D$ be a $w$-cpo. A property $P$ (a subset of $D$) is $w$-inductive (admissible) iff whenever $\langle x_n \rangle_{n \in \omega}$ is an increasing sequence of elements in $P$, then $\bigsqcup_{n \in \omega} x_n$ is also in $P$, that is,

$$(\forall n \in \omega. P(x_n)) \Rightarrow P\left(\bigsqcup_{n \in \omega} x_n\right).$$
Induction
Source Code

All the source code have been tested with Agda 2.3.2, Coq 8.4pl3 and Isabelle 2013-2.
The principle of mathematical induction

Let $A(x)$ be a propositional function. To prove $A(x)$ for all $x \in \mathbb{N}$, it suffices prove:

- the **basis** $A(0)$ and
- the **induction step**, that $A(n) \Rightarrow A(n + 1)$, for all $n \in \mathbb{N}$ ($A(n)$ is called the **induction hypothesis**).
The Principle of Mathematical Induction

First-order logic version

Let $A(x)$ be a formula with free variable $x$.

For each formula $A(x)$:

$$[A(0) \land \forall x. A(x) \Rightarrow A(x + 1)] \Rightarrow \forall x. A(x)$$  
(.axiom schema of induction)
The Principle of Mathematical Induction

First-order logic version

Let $A(x)$ be a formula with free variable $x$.

For each formula $A(x)$:

$$\left[ A(0) \land \forall x. A(x) \Rightarrow A(x + 1) \right] \Rightarrow \forall x. A(x) \quad \text{(axiom schema of induction)}$$

Equivalent formulations

$$A(0) \Rightarrow \left[ (\forall x. A(x) \Rightarrow A(x + 1)) \Rightarrow \forall x. A(x) \right] \quad \text{(by exportation)}$$

$$A(0) \Rightarrow (\forall x. A(x) \Rightarrow A(x + 1)) \Rightarrow \forall x. A(x) \quad \text{(right-assoc. conditional)}$$
The Principle of Mathematical Induction

First-order logic version

Let $A(x)$ be a formula with free variable $x$.

For each formula $A(x)$:

$$[A(0) \land \forall x. A(x) \Rightarrow A(x + 1)] \Rightarrow \forall x. A(x) \quad \text{(axiom schema of induction)}$$

Equivalent formulations

$$A(0) \Rightarrow [(\forall x. A(x) \Rightarrow A(x + 1)) \Rightarrow \forall x. A(x)] \quad \text{(by exportation)}$$

$$A(0) \Rightarrow (\forall x. A(x) \Rightarrow A(x + 1)) \Rightarrow \forall x. A(x) \quad \text{(right-assoc. conditional)}$$

Inference rule style

$$\begin{array}{c}
A(0) \\
\forall x. A(x) \Rightarrow A(x + 1)
\end{array} \quad \Rightarrow \quad \forall x. A(x)$$
Higher-order logic

“The adjective ‘first-order’ is used to distinguish the languages ... from those in which are predicates having other predicates or functions as arguments, or quantification over functions or predicates, or both.”

The Principle of Mathematical Induction

Higher-order logic

“The adjective ‘first-order’ is used to distinguish the languages ... from those in which are predicates having other predicates or functions as arguments, or quantification over functions or predicates, or both.”*

Second-order logic version

Let $X$ be a predicate variable.

\[
\forall X. X(0) \Rightarrow (\forall x. X(x) \Rightarrow X(x + 1)) \Rightarrow \forall x. X(x) \quad \text{(axiom of induction)}
\]

The Principle of Mathematical Induction

Historical remark
Peano (1889)* and Dedekind (1888)† axiom: $1 \in \mathbb{N}$

The Principle of Mathematical Induction

**Coq** generates the induction principles associated to the inductively defined (data) types.

**Example (inductive data type for the natural numbers)**

(From **Coq** standard library)

```
Inductive nat : Set :=
 | O  : nat
 | S  : nat → nat.
```
Example (cont.)

The Check `nat_ind` command yields:

```
nat_ind
  : ∀ P : nat → Prop,
  P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```
The Principle of Mathematical Induction

Example (cont.)

The Check `nat_ind` command yields:

```coq
nat_ind
  : ∀ P : nat → Prop,
  P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```

The Check `nat_rec` command yields:

```coq
nat_rec
  : ∀ P : nat → Set,
  P 0 → (∀ n : nat, P n → P (S n)) → ∀ n : nat, P n
```
The Principle of Mathematical Induction

Implementation remark

What happen if instead of using

\[\text{Inductive}\ \text{nat} : \text{Set} := 0 : \text{nat} \mid S : \text{nat} \rightarrow \text{nat}\]

we renamed the data type nat by

\[\text{Inductive}\ \text{P} : \text{Set} := 0 : \text{P} \mid S : \text{P} \rightarrow \text{P}\]

or we renamed the data constructor \(S\) by

\[\text{Inductive}\ \text{nat} : \text{Set} := 0 : \text{nat} \mid P : \text{nat} \rightarrow \text{nat}\]

?*

______________________________

*Conor McBride and James McKinna (2004). Functional Pearl: I am not a Number—I am a Free Variable.
Isabelle also generates the induction principles associated to the inductively defined (data) types.

Example (inductive data type for the natural numbers)

```
datatype nat = Z | S nat
```
Isabelle also generates the induction principles associated to the inductively defined (data) types.

Example (inductive data type for the natural numbers)

```plaintext
datatype nat = Z | S nat
```

The `print_theorems` command yields (among others):

```plaintext
nat.induct: \( \forall P \, Z \, \forall \text{nat}. \, P \, \text{nat} \Rightarrow P \, (S \, \text{nat}) \) \Rightarrow P \, ?\text{nat}
```
Agda doesn’t generate the induction principles, but the user can use pattern matching on the inductively defined (data) types.

Example (inductive data type for the natural numbers)

```agda
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```
In Agda, Coq and Isabelle, the “axiom of induction” is not an axiom.
The Principle of Mathematical Induction

In Agda, Coq and Isabelle, the “axiom of induction” is not an axiom (the introduction rules induce the induction principles).
The Principle of Mathematical Induction

In Agda, Coq and Isabelle, the “axiom of induction” is not an axiom (the introduction rules induce the induction principles).

Example (proof of the principle of mathematical induction in Agda)

\[
\mathbb{N}\text{-ind} : (A : \mathbb{N} \to \text{Set}) \to \\
A \text{ zero } \to \\
(\forall n \to A\ n \to A\ (\text{succ}\ n)) \to \\
\forall n \to A\ n
\]

\[
\mathbb{N}\text{-ind} A\ A0\ h\ \text{zero} = A0 \\
\mathbb{N}\text{-ind} A\ A0\ h\ (\text{succ}\ n) = h\ n\ (\mathbb{N}\text{-ind} A\ A0\ h\ n)
\]
Course-of-Values Induction

Course-of-values induction (strong or complete induction)

Let $A(x)$ be a propositional function. To prove $A(x)$ for all $x \in \mathbb{N}$, it suffices prove:

$$(\forall 0 \leq k < n. A(k)) \Rightarrow A(n), \text{ for all } n \in \mathbb{N}.$$
Course-of-Values Induction

Example

The Fibonacci numbers are defined by $F_0 = 0$, $F_1 = 1$ and $F_{k+2} = F_k + F_{k+1}$, so $F = \{0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots \}$.

Example

The Fibonacci numbers are defined by $F_0 = 0$, $F_1 = 1$ and $F_{k+2} = F_k + F_{k+1}$, so $F = \{0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots \}$.

Let $\Phi$ and $\hat{\Phi}$ be the roots of the equation $x^2 - x - 1$:

$$\Phi = \frac{1 + \sqrt{5}}{2} \text{ and } \hat{\Phi} = \frac{1 - \sqrt{5}}{2},$$

so $\Phi^2 = \Phi + 1$ and $\hat{\Phi}^2 = \hat{\Phi} + 1$. Then*

$$F_k = \frac{1}{\sqrt{5}}(\Phi^k - \hat{\Phi}^k), \text{ for all } k \in \mathbb{N}.$$

---

Theorem

Mathematical induction and course-of-values induction are equivalent. *

Structural induction

Let $A(X)$ be a propositional function about the structures $X$ that are defined by some recursive/inductive definition.

Structural Induction

Structural induction

Let \( A(X) \) be a propositional function about the structures \( X \) that are defined by some recursive/inductive definition.

To prove \( A(X) \) for all the structures \( X \), it suffices prove:*  

- \( A(X) \) for the basis structure(s) of \( X \) and

Structural induction

Let \( A(X) \) be a propositional function about the structures \( X \) that are defined by some recursive/inductive definition.

To prove \( A(X) \) for all the structures \( X \), it suffices prove:

- \( A(X) \) for the basis structure(s) of \( X \) and
- given a structure \( X \) whose recursive/inductive definition says is formed from \( Y_1, \ldots, Y_k \), that \( A(X) \) assuming that the properties \( A(Y_1), \ldots, A(Y_k) \) hold.

---

Example (Coq version)

The parametric inductive data type (from the standard library):

```
Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A → list A → list A.
```
Structural Induction for Lists

Example (Coq version)

The parametric inductive data type (from the standard library):

```coq
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A.
```

The induction principle:

```coq
list_ind
  : ∀ (A : Type) (P : list A → Prop),
    P (nil A) →
  (∀ (a : A) (l : list A), P l → P (cons A a l)) →
  ∀ l : list A, P l
```
Example (Isabelle version)
The polymorphic inductive data type:

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

Structural Induction for Lists

Example (Isabelle version)

The polymorphic inductive data type:

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

The induction principle:

```
list.induct: [?P Nil; ∀a list. ?P list ⇒ ?P (Cons a list)] ⇒ ?P ?list
```
Structural Induction for Lists

Example (Agda version)

The parametric inductive data type:

```
data List (A : Set) : Set where
  [] : List A
  _∷_ : A → List A → List A
```
Example (Agda version)

The parametric inductive data type:

```agda
data List (A : Set) : Set where
  [] : List A
  _∷_ : A → List A → List A
```

The induction principle:

```agda
List-ind : {A : Set} (B : List A → Set) →
  B [] →
  ((x : A) (xs : List A) → B xs → B (x ∷ xs)) →
  ∀ xs → B xs
List-ind B B[] h [] = B[]
List-ind B B[] h (x ∷ xs) = h x xs (List-ind B B[] h xs)
```
Definition (Well-founded relation)

Let $\prec$ be a binary relation on a set $A$. The relation $\prec$ is a well-founded relation if every non-empty subset $S \subseteq A$ has a minimal element, that is,

$$\forall S \subseteq A. S \neq \emptyset \Rightarrow \exists m \in S. \forall s \in S. s \not\prec m.$$
Definition (Well-founded relation)
Let $\prec$ be a binary relation on a set $A$. The relation $\prec$ is a well-founded relation if every non-empty subset $S \subseteq A$ has a minimal element, that is,

$$
\forall S \subseteq A. S \neq \emptyset \Rightarrow \exists m \in S. \forall s \in S. s \not\prec m.
$$

Definition (Well-founded induction)
Let $\prec$ be a well-founded relation on a set $A$ and $A(x)$ a propositional function. To prove $A(x)$ for all $a \in A$, it suffices prove:

$$
(\forall b \prec a. A(b)) \Rightarrow A(a), \text{ for all } a \in A.
$$
Example

Let $\prec$ be the well-founded relation on $\mathbb{N}$ given by the graph of the successor function $n \mapsto n + 1$.

Well-Founded Induction

Example

Let $\prec$ be the well-founded relation on $\mathbb{N}$ given by the graph of the successor function $n \mapsto n + 1$.

Then mathematical induction is a special case of well-founded induction.

Well-Founded Induction

Example
Let \( \prec \) be the well-founded relation on \( \mathbb{N} \) given by the graph of the successor function \( n \mapsto n + 1 \).

Then mathematical induction is a special case of well-founded induction.

Example
Let \( \prec \) be the well-founded relation “less than” on \( \mathbb{N} \).

---

Well-Founded Induction

Example
Let $\prec$ be the well-founded relation on $\mathbb{N}$ given by the graph of the successor function $n \mapsto n + 1$.

Then mathematical induction is a special case of well-founded induction.

Example
Let $\prec$ be the well-founded relation “less than” on $\mathbb{N}$.

Then course-of-values induction is a special case of well-founded induction.

Well-Founded Induction

Example
Let $\prec$ be the well-founded relation on $\mathbb{N}$ given by the graph of the successor function $n \mapsto n + 1$.

Then mathematical induction is a special case of well-founded induction.

Example
Let $\prec$ be the well-founded relation “less than” on $\mathbb{N}$.

Then course-of-values induction is a special case of well-founded induction.

Example
“If we take $\prec$ to be the relation between expressions such that $a \prec b$ holds iff $a$ is an immediate sub-expression of $b$ we obtain the principle of structural induction as a special case of well-founded induction.”

In type theory $a : A$ denotes that $a$ is a term (or proof term) of type $A$.

Empty Type

In type theory \( a : A \) denotes that \( a \) is a term (or proof term) of type \( A \).

Under the proposition-as-types principle, the empty type represents the false (absurdity or contradiction) proposition.*

Empty Type

In type theory \( a : A \) denotes that \( a \) is a term (or proof term) of type \( A \).

Under the proposition-as-types principle, the empty type represents the false (absurdity or contradiction) proposition.*

Therefore \( e : \text{EmptyType} \) represents a contradiction in our formalisation.

Empty Type

Example (Agda version)

```
data ⊥ : Set where

⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()  -- The absurd pattern.
```
Empty Type

Example (Coq version)

(From the standard library)

\textbf{Inductive} Empty\_set : \textbf{Set} :=.

Empty\_set\_rect

\quad : \forall (P : \text{Empty\_set} \to \textbf{Type}) (e : \text{Empty\_set}), P \; e
Example (Coq version)

(From the standard library)

**Inductive** Empty_set : Set :=.

Empty_set_rect

: ∀ (P : Empty_set → Type) (e : Empty_set), P e

**Theorem** emptySetElim {A : Set}(e : Empty_set) : A.

apply (Empty_set_rect (fun _ => A) e).

Qed.
Example (Coq version)

(From the standard library)

\textbf{Inductive} \texttt{Empty_set} : \texttt{Set} :=.

\texttt{Empty_set_rect}
\hspace{1em} : \forall (P : \texttt{Empty_set} \to \texttt{Type}) (e : \texttt{Empty_set}), P \ e

\textbf{Theorem} emptySetElim \{A : \texttt{Set}\}(e : \texttt{Empty_set}) : A.
\hspace{1em} apply (\texttt{Empty_set_rect} (\texttt{fun} \_ => A) \ e).
\hspace{2em} \textbf{Qed}.

\textbf{Theorem} emptySetElim' \{A : \texttt{Set}\}(e : \texttt{Empty_set}) : A.
\hspace{1em} elim e.
\hspace{2em} \textbf{Qed}. 

Verification of Functional Programs. Induction
Some Issues with Non-Strictly Positive Inductive Types

Infinite unfolding
See source code in the course web page.
Some Issues with Non-Strictly Positive Inductive Types

Infinite unfolding
See source code in the course web page.

Proving absurdity
See source code in the course web page.
Strictly Positive Inductive Types

The inductive types can be represented as least fixed-points of appropriated functions (functors).

Examples
Let 1 be the unity type, and $+$ and $\times$ be the operators for disjoint union and Cartesian product, respectively. Then

$$\text{Nat} = \mu X.1 + X,$$

$$\text{List } A = \mu X.1 + (A \times X).$$
Strictly Positive Inductive Types

Definition

“The occurrence of a type variable is positive iff it occurs within an even number of left hand sides of $\rightarrow$-types, it is strictly positive iff it never occurs on the left hand side of a $\rightarrow$-type.”*

Strictly Positive Inductive Types

**Definition**

Let $\mu X. F(X)$ be an inductive type. The type $\mu X. F(X)$ is a strictly positive type if $X$ occurs strictly positive in $F(X)$. 

<table>
<thead>
<tr>
<th>Positive types</th>
<th>Negative types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strictly positive types</td>
<td></td>
</tr>
</tbody>
</table>
**Definition**

Let $\mu X. F(X)$ be an inductive type. The type $\mu X. F(X)$ is a strictly positive type if $X$ occurs strictly positive in $F(X)$.

### Positive types

- Strictly positive types

### Negative types

**Proof assistants**

Agda, Coq and Isabelle accept only strictly positive inductive types.
Strictly Positive Inductive Types

The following examples of inductive types* are rejected by Agda (Coq and Isabelle) because they are not strictly positive inductive types.

Example (negative type)

\[ D = \mu X. X \rightarrow X \]

```haskell
data D : Set where
  lam : (D \rightarrow D) \rightarrow D
```

-- D is not strictly positive, because it occurs to the left
-- of an arrow in the type of the constructor lam in the
-- definition of D.

*Adapted from the Coq’Art, Matthes’ PhD thesis and Agda’s source code.
Example (positive, non-strictly positive type)

\[ P = \mu X. (X \rightarrow 2) \rightarrow 2 \]

```haskell
data P : Set where
    p : ((P \rightarrow \text{Bool}) \rightarrow \text{Bool}) \rightarrow P
```

-- P is not strictly positive, because it occurs to the left
-- of an arrow in the type of the constructor p in the
-- definition of P.
Co-Induction
Non-Well-Founded Sets

Axiom of foundation (ZFC)

All sets are well-founded.

---


Non-Well-Founded Sets

Axiom of foundation (ZFC)
All sets are well-founded.

Theorem
A set $X$ is well-founded, if and only if, there is no sequence $\langle X_n \mid n \in \mathbb{N} \rangle$ such that $X_0 = X$ and $X_{x+1} \in X_n$ for all $n \in \mathbb{N}$. *

Definition (Non-well-founded set)
A set $X$ is non-well-founded if there is an infinite sequence $X_1, X_2, \cdots$ such that $X_{n+1}$ is a member of $X_n$, for all $n \in \mathbb{N}$. †

Co-Inductive Types

Description

“The objects of an inductive type are well-founded with respect to the constructors of the type. In other words, such objects contain only a finite number of constructors. Co-inductive types arise from relaxing this condition, and admitting types whose objects contain an infinity of constructors.”* 

Co-Inductive Types

Description

“The objects of an inductive type are well-founded with respect to the constructors of the type. In other words, such objects contain only a finite number of constructors. Co-inductive types arise from relaxing this condition, and admitting types whose objects contain an infinity of constructors.”*

Remark

Potentially infinity of constructors.

Co-Inductive Types

Example (Haskell)

The canonical example of an co-inductive data type are streams.

```
data Stream a = Cons a (Stream a)
```

Remark

Haskell’s `data` keyword defines both inductive and co-inductive data types. That is not a good idea!
Example (Haskell)

The canonical example of a co-inductive data type are streams.

```haskell
data Stream a = Cons a (Stream a)

data Nat = Z | S Nat

zeros :: Stream Nat
zeros = Cons Z zeros
```
Example (Haskell)

The canonical example of an co-inductive data type are streams.

```haskell
data Stream a = Cons a (Stream a)

data Nat = Z | S Nat

zeros :: Stream Nat
zeros = Cons Z zeros
```

Remark

Haskell’s `data` keyword defines both inductive and co-inductive data types. That is not a good idea!
Remark

The Set Implicit Arguments command can be used in Coq for handling the implicit arguments.
Remark

The Set Implicit Arguments command can be used in Coq for handling the implicit arguments.

Example (Coq)

```
CoInductive Stream (A : Type) : Type :=
  cons : A → Stream A → Stream A.

CoFixpoint zeros : Stream nat := cons 0 zeros.
```
Co-Inductive Types

Example (cont.)

**Notation** "x :: xs" :=
(cons x xs) (at level 60, right associativity).

**CoFixpoint** zeros : Stream nat := 0 :: zeros.
Example (cont.)

Notation "x :: xs" :=
(cons x xs) (at level 60, right associativity).

CoFixpoint zeros : Stream nat := 0 :: zeros.

Remark
We will continue using Coq for the examples related to co-induction.
Co-Inductive Types

Example (co-inductive natural numbers)

Intuition: $\text{Co}\mathbb{N} = \mathbb{N} \cup \{\infty\}$

```ocaml
CoInductive Conat : Set :=
| cozero : Conat
| cosucc : Conat -> Conat.

CoFixpoint inf : Conat := cosucc inf.
```
Co-Inductive Types

Let $D$ be a set, $(D, \sqsubseteq)$ be a poset and $f$ be a function $f : D \rightarrow D$.

Definition (Post-fixed point)
An element $d \in D$ is a post-fixed point of $f$ if $d \sqsubseteq f(d)$.

Theorem
If $d \in D$ is the greatest post-fixed point of $f$, then $d$ is the greatest fixed-point of $f$.

Let $D$ be a set, $(D, \sqsubseteq)$ be a poset and $f$ be a function $f : D \rightarrow D$.

**Definition (Post-fixed point)**

An element $d \in D$ is a post-fixed point of $f$ if $d \sqsubseteq f(d)$.

**Definition (Greatest post-fixed point)**

The greatest post-fixed of $f$ is greatest among the post-fixed points of $f$. That is, $d \in D$ is the greatest post-fixed point of $f$ if:

- $d \sqsubseteq f(d)$ and
- $\forall x. x \sqsubseteq f(x) \Rightarrow x \sqsubseteq d$.

Co-Inductive Types

Let $D$ be a set, $(D, \sqsubseteq)$ be a poset and $f$ be a function $f : D \rightarrow D$.

Definition (Post-fixed point)
An element $d \in D$ is a post-fixed point of $f$ if $d \sqsubseteq f(d)$.

Definition (Greatest post-fixed point)
The greatest post-fixed of $f$ is greatest among the post-fixed points of $f$. That is, $d \in D$ is the greatest post-fixed point of $f$ if:

- $d \sqsubseteq f(d)$ and
- $\forall x. x \sqsubseteq f(x) \Rightarrow x \sqsubseteq d$.

Theorem
If $d \in D$ is the greatest post-fixed point of $f$, then $d$ is the greatest fixed-point of $f$.*

Co-Inductive Types

Representation

The inductive/co-inductive types can be represented as least/greatest fixed-points of appropriated functions (functors).

Recall that the least and greatest fixed-points of a unary function $f$ are denoted by $\mu x.f(x)$ and $\nu x.f(x)$, respectively.

Examples

Let 1 be the unity type, and $+$ and $\times$ be the operators for disjoint union and Cartesian product, respectively. Then

\[
\begin{align*}
\text{Nat} &= \mu X.1 + X, \\
\text{Conat} &= \nu X.1 + X, \\
\text{List } A &= \mu X.1 + (A \times X), \\
\text{Colist } A &= \nu X.1 + (A \times X), \\
\text{Stream } A &= \nu X.A \times X.
\end{align*}
\]
Remark

"Due to the coincidence of least and greatest fixed-point types* in lazy languages such as Haskell, the distinction between inductive and coinductive types is blurred in partial functional programming."

Definition (Recursive and co-recursive function)

Recursion function: functions from an inductive type
Co-recursive function: functions into an co-inductive type

---

Definition (Recursive and co-recursive function)

Recursion function: functions from an inductive type
Co-recursive function: functions into an co-inductive type

“…we use the term recursive program for a function whose domain is type defined recursively as the least solution of some equation.”*

Definition (Recursive and co-recursive function)

Recursion function: functions from an inductive type
Co-recursive function: functions into an co-inductive type

“...we use the term recursive program for a function whose domain is type defined recursively as the least solution of some equation.”

“...we use the term corecursive program for a function whose range is a type defined recursively as the greatest solution of some equation.”

Definition (Recursive and co-recursive function)
Recursion function: functions from an inductive type
Co-recursive function: functions into an co-inductive type

“...we use the term recursive program for a function whose domain is type defined recursively as the least solution of some equation.”*

“...we use the term corecursive program for a function whose range is a type defined recursively as the greatest solution of some equation.”*

Remark
Alternative names for co-recursion could be “non-wellfounded recursion” or “baseless recursion”.†

Co-Recursive Functions Guarded by Constructors

Condition

“Recursive calls must be protected by at least one constructor, and no other functions apart from constructors can be applied to them.”∗

Co-Recursive Functions Guarded by Constructors

Condition

“Recursive calls must be protected by at least one constructor, and no other functions apart from constructors can be applied to them.”*

Example

CoFixpoint from (n : nat) : Stream nat := n :: from (S n).

Co-Recursive Functions Guarded by Constructors

Condition

“Recursive calls must be protected by at least one constructor, and no other functions apart from constructors can be applied to them.”*

Example

```cofixpoint```
CoFixpoint from (n : nat) : Stream nat := n :: from (S n).
```cofixpoint```

Example

```cofixpoint```
CoFixpoint alter : Stream bool := true :: false :: alter.
```cofixpoint```

Example (counterexample)

CoFixpoint

filter (A : Type)(P : A → bool)(xs : Stream A) : Stream A :=
match xs with x' :: xs' =>
  if P x' then x' :: filter P xs' else filter P xs'
end.

The filter function is not guarded by constructors because there is not
constructor to guard the recursive call in the else branch.
Auxiliary definition

**Definition** tail (A : Type)(xs : Stream A) : Stream A :=
match xs with _ :: xs' => xs' end.
Co-Recursive Functions Guarded by Constructors

Auxiliary definition

Definition tail (A : Type)(xs : Stream A) : Stream A :=
match xs with _ ∷ xs' => xs' end.

Example (counterexample)

CoFixpoint zeros : Stream nat := 0 :: tail zeros.

The zeros function is not guarded by constructors because there is a function (tail) applied to the recursive call which is not a constructor.
Co-Recursive Functions Guarded by Constructors

Example

From nat to Conat (recursive version).

```coq
Fixpoint nat2conat (n : nat) : Conat :=
    match n with
    | O => cozero
    | S n' => cosucc (nat2conat n')
    end.
```

From nat to Conat (co-recursive version).

```coq
CoFixpoint nat2conat (n : nat) : Conat :=
    match n with
    | O => cozero
    | S n' => cosucc (nat2conat n')
    end.
```
Example

From nat to Conat (recursive version).

\[
\text{Fixpoint } \text{nat2conat} \ (n : \text{nat}) : \text{Conat} := \\
\begin{align*}
\text{match } n \text{ with} \\
|\ 0 & \Rightarrow \text{cozero} \\
|\ \text{S } n' & \Rightarrow \text{cosucc (nat2conat } n') \\
\text{end}.
\end{align*}
\]

From nat to Conat (co-recursive version).

\[
\text{CoFixpoint } \text{nat2conat} \ (n : \text{nat}) : \text{Conat} := \\
\begin{align*}
\text{match } n \text{ with} \\
|\ 0 & \Rightarrow \text{cozero} \\
|\ \text{S } n' & \Rightarrow \text{cosucc (nat2conat } n') \\
\text{end}.
\end{align*}
\]
Suitable notions of equality between potentially infinite terms can be defined as binary co-inductive relations.
Suitable notions of equality between potentially infinite terms can be defined as binary co-inductive relations.

**Auxiliary definition**

```coq
Definition head (A : Type)(xs : Stream A) : A :=
  match xs with x' :: _ => x' end.
```
Equality

Example (equality on streams)

The equality between streams is defined by the co-inductive bisimilarity relation.*

\[
\textbf{CoInductive} \quad \text{EqStream (A : Type)} : \text{Stream A } \rightarrow \text{Stream A } \rightarrow \text{Prop} := \\
\quad \text{eqS} : \forall \text{xs ys : Stream A,} \\
\quad \quad \text{head xs = head ys} \rightarrow \\
\quad \quad \text{EqStream (tail xs) (tail ys)} \rightarrow \\
\quad \quad \text{EqStream xs ys.}
\]

Equality

Example (equality on streams)

The equality between streams is defined by the co-inductive bisimilarity relation.*

\[
\text{CoInductive } \text{EqStream} \ (A : \text{Type}) : \text{Stream } A \to \text{Stream } A \to \text{Prop} := \\
\quad \text{eqS} : \forall \ x : \text{Stream } A, \\
\quad \quad \text{head } x = \text{head } y \rightarrow \\
\quad \quad \text{EqStream } (\text{tail } x) \ (\text{tail } y) \rightarrow \\
\quad \quad \text{EqStream } x \ y.
\]

Notation "\(x \approx y\)" :=

\((\text{EqStream } x \ y) \text{ (at level 70, no associativity)}).\]

Co-Induction Principle

Co-induction principle, greatest fixed-point induction or Park’s rule
Let $F(X)$ be a functor, then

$$\forall X. X \sqsubseteq F(X) \Rightarrow X \sqsubseteq \nu X. F(X)$$

is the co-induction principle associated to $F(X)$.*

Co-Induction Principle

Example (co-induction principle associated to $\approx$)

The functor (bisimulation):

$$F(X, xs, ys) \overset{\text{def}}{=} \text{head } xs = \text{head } ys \land X(\text{tail } xs, \text{tail } ys)$$
Co-Induction Principle

Example (co-induction principle associated to $\approx$)

The functor (bisimulation):

$$F(X, xs, ys) \overset{\text{def}}{=} \text{head } xs = \text{head } ys \land X(\text{tail } xs, \text{tail } ys)$$

The co-induction principle:

$$\forall X.(\forall xs \ ys. X(xs, ys) \Rightarrow F(X, xs, ys)) \Rightarrow$$
$$\forall xs \ ys. X(xs, ys) \Rightarrow \nu X. F(X, xs, ys)$$
Co-Induction Principle

Example (co-induction principle associated to $\approx$)
The functor (bisimulation):

$$F(X, xs, ys) \overset{\text{def}}{=} \text{head } xs = \text{head } ys \land X(\text{tail } xs, \text{tail } ys)$$

The co-induction principle:

$$\forall X. (\forall xs ys. X(xs, ys) \Rightarrow F(X, xs, ys)) \Rightarrow \forall xs ys. X(xs, ys) \Rightarrow \nu X. F(X, xs, ys)$$

The Coq type:

\[
\text{co\_ind} : \forall A : \textbf{Type}, \forall R : \text{Stream } A \to \text{Stream } A \to \textbf{Prop}, \\
(\forall xs ys : \text{Stream } A, R xs ys \rightarrow \\
\text{head } xs = \text{head } ys \land R (\text{tail } xs) (\text{tail } ys)) \rightarrow \\
\forall xs ys : \text{Stream } A, R xs ys \rightarrow xs \approx ys
\]
Co-Induction Principle

Example (the map-iterate property)

The property states that\*\†

\[
\text{map } f \ (\text{iterate } f \ x) \approx \text{iterate } f \ (f \ x).
\]

where


Example (the map-iterate property)

The property states that*:†

\[ \text{map } f \ (\text{iterate } f \ x) \approx \text{iterate } f \ (f \ x). \]

where

\textbf{CoFixpoint} \\
\text{map } (A \ B : \textbf{Type})(f : A \rightarrow B)(xs : \text{Stream } A) : \text{Stream } B := \\
\text{match } xs \ \text{with } x' :: xs' => f x' :: \text{map } f \ xs' \ \text{end}. \\

\textbf{CoFixpoint iterate} (A : \textbf{Type})(f : A \rightarrow A)(a : A) : \text{Stream } A := \\
a :: \text{iterate } f \ (f \ a).

Example (the map-iterate property)

The property states that*†

\[
\text{map } f \ (\text{iterate } f \ x) \approx \text{iterate } f \ (f \ x).
\]

where

\[
\text{CoFixpoint} \quad \text{map} \ (A \ B : \text{Type})(f : A \rightarrow B)(xs : \text{Stream } A) : \text{Stream } B := \\
\quad \text{match } xs \text{ with } x' :: xs' => f \ x' :: \text{map } f \ xs' \text{ end.}
\]

\[
\text{CoFixpoint} \quad \text{iterate} \ (A : \text{Type})(f : A \rightarrow A)(a : A) : \text{Stream } A := \\
\quad a :: \text{iterate } f \ (f \ a).
\]

See the proof in the source code in the course web page.

Zeno
Zeno (An Automated Prover for Properties of Recursive Data Structures)
Zeno (An Automated Prover for Properties of Recursive Data Structures)

- **Automatic** inductive theorem prover for proving Haskell properties
Zeno (An Automated Prover for Properties of Recursive Data Structures)

- **Automatic** inductive theorem prover for proving Haskell properties
- The tool can discover necessary auxiliary theorems
Zeno (An Automated Prover for Properties of Recursive Data Structures)

- **Automatic** inductive theorem prover for proving Haskell properties
- The tool can discover necessary auxiliary theorems
- The proofs can be verified in Isabelle
Zeno (An Automated Prover for Properties of Recursive Data Structures)

- **Automatic** inductive theorem prover for proving Haskell properties
- The tool can discover necessary auxiliary theorems
- The proofs can be verified in Isabelle
- From a test suit for IsaPlanner, Zeno can prove more properties than IsaPlanner and ACL2s (ACL2 sedan)
Demo

See source code in the course web page.
Zeno

Demo
See source code in the course web page.

Presentation (slides)
Demo
See source code in the course web page.

Presentation (slides)

Limitations
Zeno works only with terminating functions and total and finite values.
Material


- Web
  http://www.haskell.org/haskellwiki/Zeno
安装（Zeno 0.2.0.1 测试与 GHC 7.0.4）

$ cabal unpack zeno

$ cd zeno-0.2.0.1

# Remove from zeno.cabal:
if impl(ghc >= 7)
  ghc-options: -with-rtsopts="-N"

$ cabal install
Installation (Zeno 0.2.0.1 tested with GHC 7.0.4)

$ cabal unpack zeno

$ cd zeno-0.2.0.1

# Remove from zeno.cabal:
if impl(ghc >= 7)
    ghc-options: -with-rtsopts="-N"

$ cabal install

Remark

For installing/using different versions of GHC the stow command in your friend (see http://www1.eafit.edu.co/asicard/tips-and-tricks.html.)
HipSpec
HipSpec (Automating Inductive Proofs of Program Properties)

HipSpec* is based on:

- Hip†
- QuickSpec‡
- Theorem provers (e.g. E, Vampire and Z3)

Automatically prove properties about Haskell programs including partial and potentially infinite values.
Hip (Haskell Inductive Prover)

- Automatically prove properties about Haskell programs including partial and potentially infinite values.
- Subset of Haskell $\rightarrow$ intermediate language $\rightarrow$ first-order logic
Automatically prove properties about Haskell programs including partial and potentially infinite values.

Subset of Haskell $\rightarrow$ intermediate language $\rightarrow$ first-order logic

Induction techniques
- Definitional equality
- Structural induction
- Scott’s fixed-point induction
- Approximation lemma
Automatically prove properties about Haskell programs including partial and potentially infinite values.

Subset of Haskell $\rightarrow$ intermediate language $\rightarrow$ first-order logic

Induction techniques
- Definitional equality
- Structural induction
- Scott’s fixed-point induction
- Approximation lemma

The higher-order (co)-induction principles are handled at the meta-level.
Hip (Haskell Inductive Prover)

- Automatically prove properties about Haskell programs including partial and potentially infinite values.
- Subset of Haskell → intermediate language → first-order logic
- Induction techniques
  - Definitional equality
  - Structural induction
  - Scott’s fixed-point induction
  - Approximation lemma
- The higher-order (co)-induction principles are handled at the meta-level.
- The first-order reasoning is handled by off-the-shelf theorem provers (E, Prover9, SPASS, Vampire and Z3).
Data type and equality

data Prop a = a :=: a

(=:=) :: a -> a -> Prop a

(=:=) = (:=:)

Example

From combinatory logic.*

Example

From combinatory logic.*

\[ k :: a \to b \to a \]
\[ k \ x \ _ = x \]

Example

From combinatory logic.*

\[
\begin{align*}
\mathsf{k} &:: a \to b \to a \\
\mathsf{k} \; x \; _{} & = x \\
\mathsf{s} &:: (a \to b \to c) \to (a \to b) \to a \to c \\
\mathsf{s} \; f \; g \; x & = f \; x \; (g \; x)
\end{align*}
\]

Example

From combinatory logic.*

\[
\begin{align*}
&k :: a \to b \to a \\
&k \ x \ _ = x \\
&s :: (a \to b \to c) \to (a \to b) \to a \to c \\
&s \ f \ g \ x = f \ x \ (g \ x) \\
&\text{id} :: a \to a \\
&\text{id} \ x = x
\end{align*}
\]

Example

From combinatory logic.*

\[ k :: a \to b \to a \]
\[ k \ x \ _ = x \]

\[ s :: (a \to b \to c) \to (a \to b) \to a \to c \]
\[ s \ f \ g \ x = f \ x \ (g \ x) \]

\[ id :: a \to a \]
\[ id \ x = x \]

\[ prop\_skk\_id :: Prop \ (a \to a) \]
\[ prop\_skk\_id = s \ k \ k =:= id \]

Example

data N = Z | S N
Example

```plaintext
data N = Z | S N
```

- Structural recursion on total and finite values

\[
P Z \quad \forall x. P x \Rightarrow P(S x)
\]

\[
\forall x. x \text{ total and finite} \Rightarrow P x
\]
Example

```
data N = Z | S N
```

- **Structural recursion on total and finite values**

\[
P Z \quad \forall x. P x \Rightarrow P(S \ x)
\]

\[
\forall x. x \text{ total and finite } \Rightarrow P x
\]

- **Structural recursion on partial and potentially infinite values**

\[
P \bot \quad P Z \quad \forall x. P x \Rightarrow P(S \ x) \quad P \text{ admissible}
\]

\[
\forall x. P x
\]
Limitations

Hip cannot use auxiliary theorems and theories.
Limitations

Hip cannot use auxiliary theorems and theories.

Installation

Hip is now developed as a part of the HipSpec, so it is not stand-alone maintained.

You can install Hip from https://github.com/asr/hip using GHC 7.6.3.