Automated Software Verification

Ahmed Bouajjani
Université Paris Diderot

Introduction

Development of our modern life

<==>

Increasing need of automated services

Introduction

Development of our modern life

<==>

Increasing need of automated services

==>

Computers systems are ubiquitous,
handling critical tasks

Introduction

Development of our modern life

<==>

Increasing need of automated services

==>

Computers systems are ubiquitous,
handling critical tasks

==>
Complex systems,
hard to get right

==>
Misbehaviors have
inacceptable consequences

Introduction

Development of our modern life

<==>

Increasing need of automated services

==>

Computers systems are ubiquitous,
handling critical tasks

==>

Complex systems,
hard to get right

==>

Misbehaviors have
inacceptable consequences

==>

Need of principled methods of their
design and verification

Introduction

Development of our modern life

<==>

Increasing need of automated services

==>

**Computers systems are ubiquitous,
handling critical tasks**

==>

**Complex systems,
hard to get right**

==>

**Misbehaviors have
inacceptable consequences**

==>

**Need of principled methods of their
design and verification**

==>

- Formal methods for modeling and specification
- Automated verification methods and techniques

Introduction

Development of our modern life

<==>

Increasing need of automated services

==>

Computers systems are ubiquitous,
handling critical tasks

==>

Complex systems,
hard to get right

==>

Misbehaviors have
inacceptable consequences

==>

Need of principled methods of their
design and verification

==>

- Formal methods for modeling and specification
- Automated verification methods and techniques

Widely used in industry:

- **Transportation, energy: safety critical systems**
- **(hard/soft)ware ind. (Intel, IBM, ARM, Microsoft, Facebook, Amazone WS, Google, etc.): multicores, cloud computing, IoT, blockchain/smart contracts, autonomous vehicles, etc.**

Formal Modeling and Specification

- Languages with precise semantics.
- Allow precise communication between
  - the user and the designer(s)
  - the designer(s) and the developper(s)
- Allow rigorous reasoning and behaviors
  - Precise meaning of behaviors
  - Precise meaning of equivalence/refinement
  - Precise meaning of correctness

Abstraction/Refinement

- The design starts at a high level of abstraction using formal models and specifications.
- Refinement steps are needed until reaching a concrete, optimized, executable code.
- Each step must be validated.
- Early detection of design errors is important! (Their repair at late stages is hard and expansive)

Systems and their Properties

- Various formalisms depending on the abstraction level, the class of systems, and their desired properties.

- Classes of programs:
  - Functional programs
  - Sequential imperative programs
  - Reactive/parallel/concurrent programs

- Properties:
  - Partial correctness (correct when it terminates)
  - Termination
  - Safety/liveness properties

Models and Specification languages

- Transformation of data:
  - Model: Function of a data domain.
  - Specification: Input-output relation.

- Transformation of memory states
  - Model: State machine.
  - Specification: Assertions on initial-final states.

- Interaction between concurrent processes
  - Model: Parallel state machines.
  - Specification: Assertions about sequences of states.

Several approaches

- Testing
  - Applicable to the executable code
  - Cannot ensure exploration of all behaviors
  - Sometimes, the only applicable approach

- Automated theorem proving
  - Uses general logic-based framework and proof systems
  - Requires ingenuity from the user (the verifier)
  - Allows to reason at different levels of abstraction

- Algorithmic verification (model checking)
  - Use abstract operational models and decision procedures
  - Less generality than Theorem Proving but more automation
  - Useful in many practical cases

Goal of this course

- Introduce to abstract reasoning about program behaviors.
- We care in this course more about program semantics and correctness than about program complexity (performances).
- Notion of formal specification, implementation, proof of correctness of an implementation w.r.t. a specification.
- Verification approaches for basic classes of programs and properties.

Contents

- Data Manipulating Programs
  - Abstract data types
  - Functional programs, recursion
  - Imperative programs, pre/post-condition reasoning
- Reactive Systems
  - Communication, concurrency
  - Model-checking
  - Abstract analysis

Theme 1: Abstract Reasoning

Lecture 1: Abstract Data Types & Recursive Functions

Ahmed Bouajjani

Paris Diderot University – Paris 7

January 2014

Data manipulation

- Programs transform data
- They implement functions between inputs and outputs
- Examples of data domains: Booleans, Characters, Integers, Reals, Strings, Lists, Trees, etc.

Data manipulation

- Programs transform data
- They implement functions between inputs and outputs
- Examples of data domains: Booleans, Characters, Integers, Reals, Strings, Lists, Trees, etc.
- A function has a *type* (domain and co-domain):

$$f : D_1 \times \cdots \times D_n \to D$$

- Examples:

$$\begin{aligned} \wedge &: Boolean \times Boolean \to Boolean \\ + &: Nat \times Nat \to Nat \\ Sort &: List[Nat] \to List[Nat] \end{aligned}$$

Data manipulation

- Programs transform data
- They implement functions between inputs and outputs
- Examples of data domains: Booleans, Characters, Integers, Reals, Strings, Lists, Trees, etc.
- A function has a *type* (domain and co-domain):

$$f : D_1 \times \cdots \times D_n \to D$$

- Examples:

$$\begin{array}{rcl} \wedge & : & Boolean \times Boolean \to Boolean \\ + & : & Nat \times Nat \to Nat \\ Sort & : & List[Nat] \to List[Nat] \end{array}$$

- Types must be given precisely. This avoids many errors.

Defining Functions

- Finite data domains: Enumeration of its values
- Example:

$$\begin{aligned} 0 \wedge 0 &= 0 \\ 0 \wedge 1 &= 0 \\ 1 \wedge 0 &= 0 \\ 1 \wedge 1 &= 1 \end{aligned}$$

Defining Functions

- Finite data domains: Enumeration of its values
- Example:

$$
\begin{aligned}
0 \wedge 0 &= 0 \\
0 \wedge 1 &= 0 \\
1 \wedge 0 &= 0 \\
1 \wedge 1 &= 1
\end{aligned}
$$

- Not always practical, but possible in theory
- A more compact definition using a conditional construct:
  $x \wedge y = (\texttt{if } x = 0 \texttt{ then } 0 \texttt{ else } y)$

Defining Functions

- Finite data domains: Enumeration of its values
- Example:

$$\begin{aligned} 0 \wedge 0 &= 0 \\ 0 \wedge 1 &= 0 \\ 1 \wedge 0 &= 0 \\ 1 \wedge 1 &= 1 \end{aligned}$$

- Not always practical, but possible in theory
- A more compact definition using a conditional construct:
  $x \wedge y = (\texttt{if } x = 0 \texttt{ then } 0 \texttt{ else } y)$
- How to write functions over infinite domains ?

Defining Functions

- Finite data domains: Enumeration of its values
- Example:

$$\begin{aligned} 0 \wedge 0 &= 0 \\ 0 \wedge 1 &= 0 \\ 1 \wedge 0 &= 0 \\ 1 \wedge 1 &= 1 \end{aligned}$$

- Not always practical, but possible in theory
- A more compact definition using a conditional construct: $x \wedge y = (\texttt{if } x = 0 \texttt{ then } 0 \texttt{ else } y)$
- How to write functions over infinite domains ?
- We need more powerful constructs
- We need to give a structure to infinite data domains

Inductive Definition of (Potentially Infinite) Sets

- An element (object) is either basic or constructed from other objects.

Inductive Definition of (Potentially Infinite) Sets

- An element (object) is either basic or constructed from other objects.
- A set is defined by a set of constants and a set of constructors
- Example: The set *Nat* of natural numbers
  - Constant:
    $$0 : Nat$$
  - Constructor:
    $$s : Nat \rightarrow Nat$$

Inductive Definition of (Potentially Infinite) Sets

- An element (object) is either basic or constructed from other objects.
- A set is defined by a set of constants and a set of constructors
- Example: The set *Nat* of natural numbers
  - Constant:

$$0 : Nat$$

  - Constructor:

$$s : Nat \to Nat$$

- Example of elements of *Nat*:

$$0, s(0), s(s(0)), s(s(s(0))), ...$$

- Notation: n abbreviates $s^n(0)$

The General Schema

- Given a set of constants $C = \{c_1, \cdots, c_m\}$,
- Given a set of constructors of the form $\alpha : D^n \times A \to D$
- The set of element of D is the smallest set such that:
  - $C \subseteq D$
  - For every constructor $\alpha : D^n \times A \to D$, for every $d_1, \ldots, d_n \in D$, and every $a \in A$, $\alpha(d_1, \ldots, d_n, a) \in D$

The Domain of Lists

- Examples of lists:
  - $[2; 5; 8; 5]$ list of natural numbers
  - $[p; a; r; i; s]$ list of characters
  - $[[0; 2]; [2; 5; 2; 0]]$ list of lists of natural numbers

The Domain of Lists

- Examples of lists:
  - $[2;5;8;5]$ list of natural numbers
  - $[p;a;r;i;s]$ list of characters
  - $[[0;2];[2;5;2;0]]$ list of lists of natural numbers
- The domain $List[\star]$ parametrized by a domain $\star$:
  - Constant:

$$[] : List[\star]$$

  - Left-concatenation:

$$\cdot\ :\ \star \times List[\star] \to List[\star]$$

The Domain of Lists

- Examples of lists:
  - $[2; 5; 8; 5]$ list of natural numbers
  - $[p; a; r; i; s]$ list of characters
  - $[[0; 2]; [2; 5; 2; 0]]$ list of lists of natural numbers
- The domain $List[\star]$ parametrized by a domain $\star$:
  - Constant:

$$[] : List[\star]$$

  - Left-concatenation:

$$\cdot \; : \; \star \times List[\star] \rightarrow List[\star]$$

- Examples:
  - $0 \cdot [] = [0]$
  - $2 \cdot (5 \cdot (8 \cdot (5 \cdot []))) = 2 \cdot 5 \cdot 8 \cdot 5 \cdot [] = [2; 5; 8; 5]$
  - $(0 \cdot []) \cdot [] = [[0]]$
  - $[] \cdot [] = [[]] \neq []$
  - $(0 \cdot []) \cdot ((2 \cdot []) \cdot []) = [[0]; [2]]$

Defining functions over inductively defined sets

Let $f : Nat \rightarrow D$. Define $f(x)$, for every $x \in Nat$.

- Case spitting using the structure of the elements
  - $f(0) = ?$
  - $f(s(x)) = ?$

Defining functions over inductively defined sets

Let $f : Nat \to D$. Define $f(x)$, for every $x \in Nat$.

- Case spitting using the structure of the elements
  - $f(0) = ?$
  - $f(s(x)) = ?$
- Inductive definition (Recursion)

  Define $f(s(x))$ assuming that we know how to compute $f(x)$

Defining functions over inductively defined sets

Let $f : Nat \rightarrow D$. Define $f(x)$, for every $x \in Nat$.

- Case spitting using the structure of the elements
  - $f(0) = ?$
  - $f(s(x)) = ?$
- Inductive definition (Recursion)

  Define $f(s(x))$ assuming that we know how to compute $f(x)$
- Similar to proofs using structural induction

  Prove $P(0)$, and prove that $P(s(x))$ holds assuming $P(x)$.

Recursion: An Example

- Addition $+ : Nat \times Nat \rightarrow Nat$

Recursion: An Example

- Addition $+ : Nat \times Nat \to Nat$
- Recursive definition

$$\begin{aligned} 0 + x &= x \\ s(x_1) + x_2 &= s(x_1 + x_2) \end{aligned}$$

Recursion: An Example

- Addition $+ : Nat \times Nat \to Nat$
- Recursive definition

$$\begin{aligned} 0 + x &= x \\ s(x_1) + x_2 &= s(x_1 + x_2) \end{aligned}$$

- Computation

$$\begin{aligned} s(s(0)) + s(0) &= s(s(0) + s(0)) \\ &= s(s(0 + s(0))) \\ &= s(s(s(0))) \end{aligned}$$

Recursion: Another Example

- Append function $@ : List[\star] \times List[\star] \rightarrow List[\star]$
- Example: $[2; 5; 7]@[1; 5] = [2; 5; 7; 1; 5]$

Recursion: Another Example

- Append function $\texttt{@} : \mathit{List}[\star] \times \mathit{List}[\star] \to \mathit{List}[\star]$
- Example: $[2; 5; 7]\texttt{@}[1; 5] = [2; 5; 7; 1; 5]$
- Recursive definition

$$\begin{aligned} []\texttt{@}\ell &= \\ (a \cdot \ell_1)\texttt{@}\ell_2 &= \end{aligned}$$

Recursion: Another Example

- Append function $\mathtt{@} : List[\star] \times List[\star] \rightarrow List[\star]$
- Example: $[2; 5; 7]\mathtt{@}[1; 5] = [2; 5; 7; 1; 5]$
- Recursive definition

$$
\begin{aligned}
[]\mathtt{@}\ell &= \ell \\
(a \cdot \ell_1)\mathtt{@}\ell_2 &=
\end{aligned}
$$

Recursion: Another Example

- Append function $@ : List[\star] \times List[\star] \rightarrow List[\star]$
- Example: $[2; 5; 7]@[1; 5] = [2; 5; 7; 1; 5]$
- Recursive definition

$$
\begin{array}{rcl}
[]@\ell & = & \ell \\
(a \cdot \ell_1)@\ell_2 & = & a \cdot (\ell_1 @ \ell_2)
\end{array}
$$

Recursion: Another Example

- Append function $\texttt{@} : List[\star] \times List[\star] \rightarrow List[\star]$
- Example: $[2;5;7]\texttt{@}[1;5] = [2;5;7;1;5]$
- Recursive definition

$$\begin{array}{rcl} []\texttt{@}\ell & = & \ell \\ (a \cdot \ell_1)\texttt{@}\ell_2 & = & a \cdot (\ell_1 \texttt{@} \ell_2) \end{array}$$

- Computation:

$$\begin{array}{rcl} (2 \cdot 5 \cdot 7 \cdot [])\texttt{@}(1 \cdot 5 \cdot []) & = & 2 \cdot ((5 \cdot 7 \cdot [])\texttt{@}(1 \cdot 5 \cdot [])) \\ & = & 2 \cdot 5 \cdot ((7 \cdot [])\texttt{@}(1 \cdot 5 \cdot [])) \\ & = & 2 \cdot 5 \cdot 7 \cdot ([]\texttt{@}(1 \cdot 5 \cdot [])) \\ & = & 2 \cdot 5 \cdot 7 \cdot 1 \cdot 5 \cdot [] \end{array}$$

Composition: Functions can call other functions

- Multiplication $* : Nat \times Nat \rightarrow Nat$

Composition: Functions can call other functions

- Multiplication $\quad * : Nat \times Nat \rightarrow Nat$
- Recursive definition

$$
\begin{aligned}
0 * x &= \\
s(x_1) * x_2 &=
\end{aligned}
$$

Composition: Functions can call other functions

- Multiplication $\quad * : Nat \times Nat \rightarrow Nat$
- Recursive definition

$$
\begin{aligned}
0 * x &= 0 \\
s(x_1) * x_2 &=
\end{aligned}
$$

Composition: Functions can call other functions

- Multiplication $\quad * : Nat \times Nat \rightarrow Nat$
- Recursive definition

$$\begin{aligned} 0 * x &= 0 \\ s(x_1) * x_2 &= (x_1 * x_2) + x_2 \end{aligned}$$

Composition: Functions can call other functions

- Multiplication $\quad * : Nat \times Nat \to Nat$
- Recursive definition

$$\begin{aligned} 0 * x &= 0 \\ s(x_1) * x_2 &= (x_1 * x_2) + x_2 \end{aligned}$$

- Computation

$$\begin{aligned} s^2(0) * s^3(0) &= (s(0) * s^3(0)) + s^3(0) \\ &= ((0 * s^3(0)) + s^3(0)) + s^3(0) \\ &= (0 + s^3(0)) + s^3(0) \\ &= s^3(0) + s^3(0) = s(s^2(0)) + s^3(0) \\ &= s(s^2(0) + s^3(0)) \\ &= s(s(s(0) + s^3(0))) \\ &= s(s(s(0 + s^3(0)))) \\ &= s(s(s(s^3(0)))) = s^6(0) \end{aligned}$$

Composition: Another Example

- Factorial function $fact : Nat \rightarrow Nat$

Composition: Another Example

- Factorial function $fact : Nat \to Nat$
- Recursive definition

$$\begin{aligned} fact(0) &= \\ fact(s(x)) &= \end{aligned}$$

Composition: Another Example

- Factorial function $\quad fact : Nat \rightarrow Nat$
- Recursive definition

$$\begin{aligned} fact(0) &= s(0) \\ fact(s(x)) &= \end{aligned}$$

Composition: Another Example

- Factorial function $fact : Nat \rightarrow Nat$
- Recursive definition

$$
\begin{aligned}
fact(0) &= s(0) \\
fact(s(x)) &= s(x) * fact(x)
\end{aligned}
$$

Composition: Another Example

- Factorial function $fact : Nat \rightarrow Nat$
- Recursive definition

$$
\begin{aligned}
fact(0) &= s(0) \\
fact(s(x)) &= s(x) * fact(x)
\end{aligned}
$$

- Computation

$$
\begin{aligned}
fact(s(s(0))) &= s(s(0)) * fact(s(0)) \\
&= s(s(0)) * (s(0) * fact(0)) \\
&= s(s(0)) * (s(0) * s(0)) \\
&= s(0) * (s(0) * s(0)) + s(0) * s(0) \\
&= 0 * (s(0) * s(0)) + s(0) * s(0) + s(0) * s(0) \\
&= \ldots
\end{aligned}
$$

Composition: Yet Another Example

- Reverse function $Rev : List[\star] \rightarrow List[\star]$
- Example: $Rev([2; 5; 2; 1]) = [1; 2; 5; 2]$

Composition: Yet Another Example

- Reverse function $\quad Rev : List[\star] \to List[\star]$
- Example: $Rev([2;5;2;1]) = [1;2;5;2]$
- Recursive definition:

$$\begin{aligned} Rev([]) &= \\ Rev(a \cdot \ell) &= \end{aligned}$$

Composition: Yet Another Example

- Reverse function $Rev : List[\star] \rightarrow List[\star]$
- Example: $Rev([2;5;2;1]) = [1;2;5;2]$
- Recursive definition:

$$\begin{aligned} Rev([]) &= [] \\ Rev(a \cdot \ell) &= \end{aligned}$$

Composition: Yet Another Example

- Reverse function $Rev : List[\star] \to List[\star]$
- Example: $Rev([2; 5; 2; 1]) = [1; 2; 5; 2]$
- Recursive definition:

$$\begin{array}{rcl} Rev([]) & = & [] \\ Rev(a \cdot \ell) & = & Rev(\ell) @ [a] \end{array}$$

Composition: Yet Another Example

- Reverse function $Rev : List[\star] \to List[\star]$
- Example: $Rev([2;5;2;1]) = [1;2;5;2]$
- Recursive definition:

$$\begin{array}{rcl} Rev([]) & = & [] \\ Rev(a \cdot \ell) & = & Rev(\ell)@[a] \end{array}$$

- Computation

$$\begin{array}{rcl} Rev([2;5;1]) & = & Rev([5;1])@[2] \\ & = & (Rev([1])@[5])@[2] \\ & = & ((Rev([])@[1])@[5])@[2] \\ & = & ([1]@[5])@[2] \\ & = & [1;5]@[2] \\ & \ldots & \\ & = & [1;5;2] \end{array}$$

Functions between different domains

- The Length function $|\cdot| \ : \ List[\star] \rightarrow Nat$

Functions between different domains

- The Length function $|\cdot| \; : \; \mathit{List}[\star] \rightarrow \mathit{Nat}$

$$\begin{aligned} |[]| &= 0 \\ |a \cdot \ell| &= s(|\ell|) \end{aligned}$$

Functions between different domains

- The Length function $\quad |\cdot| \;:\; List[\star] \rightarrow Nat$

$$\begin{array}{rcl} |[]| & = & 0 \\ |a \cdot \ell| & = & s(|\ell|) \end{array}$$

- Sum of the elements $\quad \Sigma \;:\; List[Nat] \rightarrow Nat$

Functions between different domains

- The Length function $|\cdot| \; : \; List[\star] \rightarrow Nat$

$$
\begin{array}{rcl}
|[]| & = & 0 \\
|a \cdot \ell| & = & s(|\ell|)
\end{array}
$$

- Sum of the elements $\Sigma \; : \; List[Nat] \rightarrow Nat$

$$
\begin{array}{rcl}
\Sigma([]) & = & 0 \\
\Sigma(n \cdot \ell) & = & n + \Sigma(\ell)
\end{array}
$$

Inductive definition of functions: A General Schema

Let $f : D \times E \to F$.

- For every constant $c \in D$ and every $e \in E$, define $f(c, e)$ (as an element of F)
- For every constructor $\alpha : D^n \times A \to D$, for every $e \in E$, define $f(\alpha(x_1, \cdots, x_n, a), e)$ using a and $f(x_1, e), \cdots, f(x_n, e)$.

Proving facts about functions

- Neutral element:

$$\forall x \in Nat.\ x * s(0) = s(0) * x = x$$

- Commutativity:

$$\forall x, y \in Nat.\ x + y = y + x$$

- Associativity:

$$\forall x, y, z \in Nat.\ x + (y + z) = (x + y) + z$$

- Distributivity:

$$\forall x, y, z \in Nat.\ x * (y + z) = (x * y) + (x * z)$$

- Idempotence:

$$\forall \ell \in List[\star].\ Rev(Rev(\ell)) = \ell$$

- Kind of distributivity:

$$\forall \ell_1, \ell_2 \in List[\star].\ Rev(\ell_1 @ \ell_2) = Rev(\ell_2) @ Rev(\ell_1)$$

Structural Induction

Let $c_1, \ldots, c_m$ be the constants, and let $\alpha_1, \ldots, \alpha_n$ be the constructors.

$$\frac{\begin{array}{c} P(c_1) \\ \\ \ldots \\ \\ P(c_m) \\ \\ (\bigwedge_{i=1}^{K_1} P(x_i)) \Rightarrow P(\alpha_1(x_1, \cdots x_{K_1})) \\ \\ \ldots \\ \\ (\bigwedge_{i=1}^{K_n} P(x_i)) \Rightarrow P(\alpha_n(x_1, \cdots x_{K_n})) \end{array}}{\forall x.\ P(x)}$$

Proving Neutrality of 1 for $*$

$$\forall x \in Nat.\ x * s(0) = s(0) * x = x$$

Proving Neutrality of 1 for $*$

$$\forall x \in Nat.\ x * s(0) = s(0) * x = x$$

- Case $x = 0$.
  - $0 * s(0) = 0$
  - $s(0) * 0 = 0 * 0 + 0 = 0 + 0 = 0$

Proving Neutrality of 1 for $*$

$$\forall x \in Nat.\ x * s(0) = s(0) * x = x$$

- Case $x = 0$.
  - $0 * s(0) = 0$
  - $s(0) * 0 = 0 * 0 + 0 = 0 + 0 = 0$
- Case $x = s(x')$. Induction Hypothesis: $x' * s(0) = s(0) * x' = x'$

Proving Neutrality of 1 for $*$

$$\forall x \in Nat.\ x * s(0) = s(0) * x = x$$

- Case $x = 0$.
  - $0 * s(0) = 0$
  - $s(0) * 0 = 0 * 0 + 0 = 0 + 0 = 0$
- Case $x = s(x')$. Induction Hypothesis: $x' * s(0) = s(0) * x' = x'$
  - $s(x')*s(0) = (x'*s(0))+s(0) = x'+s(0) = s(0)+x' = s(0+x') = s(x')$ (uses commutativity of $+$)

Proving Neutrality of 1 for $*$

$$\forall x \in Nat.\ x * s(0) = s(0) * x = x$$

- Case $x = 0$.
  - $0 * s(0) = 0$
  - $s(0) * 0 = 0 * 0 + 0 = 0 + 0 = 0$
- Case $x = s(x')$. Induction Hypothesis: $x' * s(0) = s(0) * x' = x'$
  - $s(x') * s(0) = (x' * s(0)) + s(0) = x' + s(0) = s(0) + x' = s(0 + x') = s(x')$ (uses commutativity of $+$)
  - $s(0) * s(x') = (0 * s(x')) + s(x') = 0 + s(x') = s(x')$

Proving Commutativity of +

$$\forall x, y \in Nat.\ x + y = y + x$$

Proving Commutativity of $+$

$$\forall x, y \in Nat.\ x + y = y + x$$

- Case $x = 0.\ \Rightarrow x + y = 0 + y = y$
  $\rightsquigarrow \forall y \in Nat.\ y = y + 0$?

Proving Commutativity of $+$

$$\forall x, y \in Nat.\ x + y = y + x$$

- Case $x = 0$. $\Rightarrow x + y = 0 + y = y$
  $\rightsquigarrow \forall y \in Nat.\ y = y + 0$?
  - Case $y = 0$: $y + 0 = 0 + 0 = 0$

Proving Commutativity of $+$

$$\forall x, y \in Nat.\ x + y = y + x$$

- Case $x = 0$. $\Rightarrow x + y = 0 + y = y$
  $\leadsto \forall y \in Nat.\ y = y + 0$?
  - Case $y = 0$: $y + 0 = 0 + 0 = 0$
  - Case $y = s(y')$:
    - Induction hypothesis: $y' = y' + 0$
    - $y + 0 = s(y') + 0 = s(y' + 0) = s(y') = y$

Proving Commutativity of $+$

$$\forall x, y \in Nat.\ x + y = y + x$$

- Case $x = 0$. $\Rightarrow x + y = 0 + y = y$
  $\rightsquigarrow \forall y \in Nat.\ y = y + 0$?
  - Case $y = 0$: $y + 0 = 0 + 0 = 0$
  - Case $y = s(y')$:
    - ★ Induction hypothesis: $y' = y' + 0$
    - ★ $y + 0 = s(y') + 0 = s(y' + 0) = s(y') = y$
- Case $x = s(x')$. Induction Hypothesis: $\forall z \in Nat.\ x' + z = z + x'$
  $\rightsquigarrow \forall y \in Nat.\ s(x') + y = y + s(x')$?

Proving Commutativity of $+$

$$\forall x, y \in Nat.\ x + y = y + x$$

- Case $x = 0$. $\Rightarrow x + y = 0 + y = y$
  $\rightsquigarrow \forall y \in Nat.\ y = y + 0$?
  - Case $y = 0$: $y + 0 = 0 + 0 = 0$
  - Case $y = s(y')$:
    - ★ Induction hypothesis: $y' = y' + 0$
    - ★ $y + 0 = s(y') + 0 = s(y' + 0) = s(y') = y$
- Case $x = s(x')$. Induction Hypothesis: $\forall z \in Nat.\ x' + z = z + x'$
  $\rightsquigarrow \forall y \in Nat.\ s(x') + y = y + s(x')$?
  - Case $y = 0$: $s(x') + 0 = s(x' + 0) = s(0 + x') = s(x') = 0 + s(x')$

Proving Commutativity of $+$

$$\forall x, y \in Nat.\ x + y = y + x$$

- Case $x = 0$. $\Rightarrow x + y = 0 + y = y$
  $\rightsquigarrow \forall y \in Nat.\ y = y + 0$?
  - Case $y = 0$: $y + 0 = 0 + 0 = 0$
  - Case $y = s(y')$:
    - ★ Induction hypothesis: $y' = y' + 0$
    - ★ $y + 0 = s(y') + 0 = s(y' + 0) = s(y') = y$
- Case $x = s(x')$. Induction Hypothesis: $\forall z \in Nat.\ x' + z = z + x'$
  $\rightsquigarrow \forall y \in Nat.\ s(x') + y = y + s(x')$?
  - Case $y = 0$: $s(x') + 0 = s(x' + 0) = s(0 + x') = s(x') = 0 + s(x')$
  - Case $y = s(y')$:
    - ★ Induction hypothesis: $s(x') + y' = y' + s(x')$
    - ★ $s(x') + s(y') = s(x' + s(y')) = s(s(y') + x') = s(s(y' + x'))$
    - ★ $s(y') + s(x') = s(y' + s(x')) = s(s(x') + y') = s(s(x' + y'))$
    - ★ $s(s(x' + y')) = s(s(y' + x'))$

Summary

- The first step in defining a function is to define its type (its domain and its co-domain).
- Infinite data domain can be defined inductively (set of constants and a set of constructors).
- Functions over infinite data domains by reasoning on the inductive structure of the data domains.
- Facts about recursive functions can be proved by reasoning on the inductive structure of the data domains.