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
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 unacceptable 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 unacceptable 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.

• Transformation of memory states
  – Model: State machine.

• Interaction between concurrent processes
  – Model: Parallel state machines.
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):
  \[ f : D_1 \times \cdots \times D_n \rightarrow D \]
- Examples:
  \[ \wedge : \text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean} \]
  \[ + : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \]
  \[ \text{Sort} : \text{List[Nat]} \rightarrow \text{List[Nat]} \]
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):
  \[ f : D_1 \times \cdots \times D_n \rightarrow D \]
  
- Examples:
  \[
    \land : \text{Boolean} \times \text{Boolean} \rightarrow \text{Boolean} \\
    + : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \\
    \text{Sort} : \text{List[Nat]} \rightarrow \text{List[Nat]}
  \]

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

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

\[
\begin{align*}
0 \land 0 &= 0 \\
0 \land 1 &= 0 \\
1 \land 0 &= 0 \\
1 \land 1 &= 1
\end{align*}
\]
Defining Functions

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

\[
\begin{align*}
0 \land 0 &= 0 \\
0 \land 1 &= 0 \\
1 \land 0 &= 0 \\
1 \land 1 &= 1
\end{align*}
\]

- Not always practical, but possible in theory
- A more compact definition using a conditional construct:
  \[ x \land y = (\text{if } x = 0 \text{ then } 0 \text{ else } y) \]
Defining Functions

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

\[
\begin{align*}
0 \land 0 &= 0 \\
0 \land 1 &= 0 \\
1 \land 0 &= 0 \\
1 \land 1 &= 1 \\
\end{align*}
\]

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

- How to write functions over infinite domains?
Defining Functions

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

\[
\begin{align*}
0 \land 0 &= 0 \\
0 \land 1 &= 0 \\
1 \land 0 &= 0 \\
1 \land 1 &= 1
\end{align*}
\]

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

- 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:
    - $s : Nat \rightarrow Nat$

Example of elements of $Nat$: $0, s(0), s(s(0)), \ldots$
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 $\text{Nat}$ of natural numbers
  - Constant:
    $$0 : \text{Nat}$$
  - Constructor:
    $$s : \text{Nat} \rightarrow \text{Nat}$$
- Example of elements of $\text{Nat}$:
  $$0, s(0), s(s(0)), s(s(s(0))), \ldots$$
- Notation: $n$ abbreviates $s^n(0)$
The General Schema

- Given a set of constants $C = \{c_1, \ldots, c_m\}$,
- 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:
    \(\cdot : \star \times List[\star] \rightarrow List[\star]\)
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 \(\text{List}[\star]\) parametrized by a domain \(\star\):
  - Constant:
    \[
    [] : \text{List}[\star]
    \]
  - Left-concatenation:
    \[
    \cdot : \star \times \text{List}[\star] \rightarrow \text{List}[\star]
    \]

- 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 splitting using the structure of the elements
  - $f(0) = ?$
  - $f(s(x)) = ?$
Defining functions over inductively defined sets

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

- Case splitting 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 $f : \text{Nat} \rightarrow D$. Define $f(x)$, for every $x \in \text{Nat}$.

- Case splitting 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 \( + \colon \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \)
Recursion: An Example

- Addition  \( + : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \)
- Recursive definition

\[
\begin{align*}
0 + x &= x \\
s(x_1) + x_2 &= s(x_1 + x_2)
\end{align*}
\]
Recursion: An Example

- Addition  \( + : Nat \times Nat \rightarrow Nat \)
- Recursive definition

\[
\begin{align*}
0 + x &= x \\
\textit{s}(x_1) + x_2 &= \textit{s}(x_1 + x_2)
\end{align*}
\]

- Computation

\[
\begin{align*}
\textit{s} (\textit{s}(0)) + \textit{s}(0) &= \textit{s} (\textit{s}(0) + \textit{s}(0)) \\
&= \textit{s} (\textit{s}(0 + \textit{s}(0))) \\
&= \textit{s} (\textit{s}(\textit{s}(0)))
\end{align*}
\]
Recursion: Another Example

- Append function $\odot : List[*] \times List[*] \rightarrow List[*]$

- Example: $[2; 5; 7] \odot [1; 5] = [2; 5; 7; 1; 5]$
Recursion: Another Example

- Append function \( @ : List[\ast] \times List[\ast] \rightarrow List[\ast] \)

- Example: \([2; 5; 7]@[1; 5] = [2; 5; 7; 1; 5]\)

- Recursive definition

\[
\begin{align*}
[]@l &= \\
(a \cdot l_1)@l_2 &=
\end{align*}
\]
Recursion: Another Example

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

\[
\begin{align*}
\text{[]}@[\ell] &= \ell \\
(a \cdot \ell_1)@[\ell_2] &= \\
\end{align*}
\]
Recursion: Another Example

- Append function \( \circlearrowleft : \text{List}[\ast] \times \text{List}[\ast] \rightarrow \text{List}[\ast] \)

- Example: \([2; 5; 7] \circlearrowleft [1; 5] = [2; 5; 7; 1; 5]\)

- Recursive definition

\[
\begin{align*}
[\ ] \circlearrowleft l &= l \\
(a \cdot l_1) \circlearrowleft l_2 &= a \cdot (l_1 \circlearrowleft l_2)
\end{align*}
\]
Recursion: Another Example

- Append function \( \cdot @ : List[*] \times List[*] \rightarrow List[*] \)

- Example: \([2; 5; 7]@[1; 5] = [2; 5; 7; 1; 5]\)

- Recursive definition

  \[
  \begin{align*}
  []@l &= l \\
  (a \cdot l_1)@l_2 &= a \cdot (l_1@l_2)
  \end{align*}
  \]

- Computation:

  \[
  \begin{align*}
  (2 \cdot 5 \cdot 7 \cdot [])@(1 \cdot 5 \cdot []) &= 2 \cdot ((5 \cdot 7 \cdot [])@(1 \cdot 5 \cdot [])) \\
  &= 2 \cdot 5 \cdot ((7 \cdot [])@(1 \cdot 5 \cdot [])) \\
  &= 2 \cdot 5 \cdot 7 \cdot ([]@(1 \cdot 5 \cdot [])) \\
  &= 2 \cdot 5 \cdot 7 \cdot 1 \cdot 5 \cdot []
  \end{align*}
  \]
Composition: Functions can call other functions

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

Recursive definition

$0 \times x = 0$

$s(x_1 \times x_2) = (s(x_1) \times x_2) + s(x_2)$

Computation

$s_2(0) \times s_3(0) = (s_2(0) \times s_3(0)) + s_3(0)$

$= (0 + s_3(0)) + s_3(0) = s_3(0) + s_3(0) = s_3(s_2(0)) + s_3(0) = s_3(s_2(s_3(0))) = s_6(0)$
Composition: Functions can call other functions

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

\[
0 \times x = \\
\text{s}(x_1) \times x_2 =
\]
Composition: Functions can call other functions

- Multiplication $*: \text{Nat} \times \text{Nat} \rightarrow \text{Nat}$
- Recursive definition

\[
\begin{align*}
0 \times x &= 0 \\
\text{s}(x_1) \times x_2 &= 
\end{align*}
\]
Composition: Functions can call other functions

- Multiplication \( \ast : Nat \times Nat \rightarrow Nat \)
- Recursive definition

\[
\begin{align*}
0 \ast x &= 0 \\
s(x_1) \ast x_2 &= (x_1 \ast x_2) + x_2
\end{align*}
\]
Composition: Functions can call other functions

- **Multiplication**  \( \ast : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \)
- **Recursive definition**

\[
\begin{align*}
0 \ast x &= 0 \\
\text{s}(x_1) \ast x_2 &= (x_1 \ast x_2) + x_2
\end{align*}
\]

- **Computation**

\[
\begin{align*}
\text{s}^2(0) \ast \text{s}^3(0) &= (\text{s}(0) \ast \text{s}^3(0)) + \text{s}^3(0) \\
&= ((0 \ast \text{s}^3(0)) + \text{s}^3(0)) + \text{s}^3(0) \\
&= (0 + \text{s}^3(0)) + \text{s}^3(0) \\
&= \text{s}^3(0) + \text{s}^3(0) = \text{s}(\text{s}^2(0)) + \text{s}^3(0) \\
&= \text{s}(\text{s}^2(0) + \text{s}^3(0)) \\
&= \text{s}(\text{s}(\text{s}(0) + \text{s}^3(0))) \\
&= \text{s}(\text{s}(\text{s}(\text{s}(0 + \text{s}^3(0)))))) \\
&= \text{s}(\text{s}(\text{s}(\text{s}(\text{s}^3(0))))) = \text{s}^6(0)
\end{align*}
\]
Composition: Another Example

- Factorial function \( \text{fact} : \text{Nat} \rightarrow \text{Nat} \)
Composition: Another Example

- Factorial function \( \text{fact} : \text{Nat} \rightarrow \text{Nat} \)
- Recursive definition

\[
\begin{align*}
\text{fact}(0) &= 1 \\
\text{fact}(s(x)) &= s(x) \times \text{fact}(x)
\end{align*}
\]
Composition: Another Example

- Factorial function  \( \text{fact} : \text{Nat} \rightarrow \text{Nat} \)
- Recursive definition

\[
\begin{align*}
\text{fact}(0) &= s(0) \\
\text{fact}(s(x)) &= \ldots
\end{align*}
\]
Composition: Another Example

- Factorial function \( \text{fact} : \text{Nat} \rightarrow \text{Nat} \)
- Recursive definition

\[
\begin{align*}
\text{fact}(0) &= s(0) \\
\text{fact}(s(x)) &= s(x) \ast \text{fact}(x)
\end{align*}
\]
Composition: Another Example

- Factorial function  \( \text{fact} : \text{Nat} \rightarrow \text{Nat} \)
- Recursive definition

\[
\begin{align*}
\text{fact}(0) &= s(0) \\
\text{fact}(s(x)) &= s(x) \times \text{fact}(x)
\end{align*}
\]

- Computation

\[
\begin{align*}
\text{fact}(s(s(0))) &= s(s(0)) \times \text{fact}(s(0)) \\
&= s(s(0)) \times (s(0) \times \text{fact}(0)) \\
&= s(s(0)) \times (s(0) \times s(0)) \\
&= s(0) \times (s(0) \times s(0)) + s(0) \times s(0) \\
&= 0 \times (s(0) \times s(0)) + s(0) \times s(0) + s(0) \times s(0) \\
&= \ldots
\end{align*}
\]
Composition: Yet Another Example

- Reverse function \( \text{Rev} : \text{List}[\star] \rightarrow \text{List}[\star] \)
- Example: \( \text{Rev}([2; 5; 2; 1]) = [1; 2; 5; 2] \)
Composition: Yet Another Example

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

  $$
  \begin{align*}
  Rev([]) &= \\
  Rev(a \cdot \ell) &=
  \end{align*}
  $$
Composition: Yet Another Example

- Reverse function \( \text{Rev} : List[*] \rightarrow List[*] \)
- Example: \( \text{Rev}([2; 5; 2; 1]) = [1; 2; 5; 2] \)
- Recursive definition:

\[
\begin{align*}
\text{Rev}([]) & = [] \\
\text{Rev}(a \cdot \ell) & = \\
\end{align*}
\]
Composition: Yet Another Example

- Reverse function \( \text{Rev} : List[\ast] \rightarrow List[\ast] \)
- Example: \( \text{Rev}([2; 5; 2; 1]) = [1; 2; 5; 2] \)
- Recursive definition:

\[
\begin{align*}
\text{Rev}([]) &= [] \\
\text{Rev}(a \cdot \ell) &= \text{Rev}(\ell)@a
\end{align*}
\]
Composition: Yet Another Example

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

\[
\begin{align*}
Rev([\,]) &= [] \\
Rev(a \cdot \ell) &= Rev(\ell)@[a]
\end{align*}
\]

- Computation

\[
\begin{align*}
Rev([2; 5; 1]) &= Rev([5; 1])@[2] \\
&= (Rev([1])@[5])@[2] \\
&= ((Rev([\,])@[1])@[5])@[2] \\
&= ([1]@[5])@[2] \\
&= [1; 5]@[2] \\
&= \ldots \\
&= [1; 5; 2]
\end{align*}
\]
Functions between different domains

- The Length function $|·| : \text{List}[*] \rightarrow \text{Nat}$
Functions between different domains

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

\[
\begin{align*}
|[]| &= 0 \\
|a \cdot \ell| &= s(|\ell|)
\end{align*}
\]
Functions between different domains

- The Length function $|·| : \text{List}[*] \rightarrow \text{Nat}$
  
  \[
  |[]| = 0 \\
  |a \cdot l| = s(|l|)
  \]

- Sum of the elements $\Sigma : \text{List}[\text{Nat}] \rightarrow \text{Nat}$
Functions between different domains

- The Length function \( | \cdot | : \text{List}[\star] \rightarrow \text{Nat} \)
  \[
  |[]| = 0 \\
  |a \cdot \ell| = s(|\ell|)
  \]

- Sum of the elements \( \Sigma : \text{List}[\text{Nat}] \rightarrow \text{Nat} \)
  \[
  \Sigma([]) = 0 \\
  \Sigma(n \cdot \ell) = n + \Sigma(\ell)
  \]
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:
  \[ \forall x \in \text{Nat}. \ x \ast s(0) = s(0) \ast x = x \]

- Commutativity:
  \[ \forall x, y \in \text{Nat}. \ x + y = y + x \]

- Associativity:
  \[ \forall x, y, z \in \text{Nat}. \ x + (y + z) = (x + y) + z \]

- Distributivity:
  \[ \forall x, y, z \in \text{Nat}. \ x \cdot (y + z) = (x \cdot y) + (x \cdot z) \]

- Idempotence:
  \[ \forall \ell \in \text{List}[\star]. \ \text{Rev}(\text{Rev}(\ell)) = \ell \]

- Kind of distributivity:
  \[ \forall \ell_1, \ell_2 \in \text{List}[\star]. \ \text{Rev}(\ell_1 \@ \ell_2) = \text{Rev}(\ell_2) \@ \text{Rev}(\ell_1) \]
Structural Induction

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

$$P(c_1)$$

$$\ldots$$

$$P(c_m)$$

$$\left( \bigwedge_{i=1}^{K_1} P(x_i) \right) \Rightarrow P(\alpha_1(x_1, \ldots x_{K_1}))$$

$$\ldots$$

$$\left( \bigwedge_{i=1}^{K_n} P(x_i) \right) \Rightarrow P(\alpha_n(x_1, \ldots x_{K_n}))$$

$$\forall x. \ P(x)$$
Proving Neutrality of 1 for \(*\)

\[ \forall x \in \text{Nat. } x * s(0) = s(0) * x = x \]
Proving Neutrality of 1 for *

\[ \forall x \in \text{Nat. } x \times s(0) = s(0) \times x = x \]

- Case \( x = 0 \).
  - \( 0 \times s(0) = 0 \)
  - \( s(0) \times 0 = 0 \times 0 + 0 = 0 + 0 = 0 \)
Proving Neutrality of $1$ for $\ast$

$$\forall x \in \text{Nat.} \ x \ast s(0) = s(0) \ast x = x$$

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

- **Case** $x = s(x')$. Induction Hypothesis: $x' \ast s(0) = s(0) \ast x' = x'$
Proving Neutrality of 1 for *

\[ \forall x \in \text{Nat}. \ x \star s(0) = s(0) \star x = x \]

- **Case** \( x = 0 \).
  - 0 \( \star \) \( s(0) = 0 \)
  - \( s(0) \star 0 = 0 \star 0 + 0 = 0 + 0 = 0 \)

- **Case** \( x = s(x') \). **Induction Hypothesis**: \( x' \star s(0) = s(0) \star x' = x' \)
  - \( s(x') \star s(0) = (x' \star s(0)) + s(0) = x' + s(0) = s(0) + x' = s(0 + x') = s(x') \)
  - (uses commutativity of +)
Proving Neutrality of 1 for $*$

$$\forall x \in \text{Nat.} \, x * s(0) = s(0) * x = x$$

- 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 $+$

$\forall x, y \in \text{Nat.} \ x + y = y + x$
Proving Commutativity of +

$$\forall x, y \in \text{Nat. } x + y = y + x$$

- Case $x = 0. \ \Rightarrow x + y = 0 + y = y$
  $\Rightarrow \forall y \in \text{Nat. } y = y + 0$
Proving Commutativity of $+$

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

- Case $x = 0$. $\Rightarrow x + y = 0 + y = y$
  $\leadsto \forall y \in Nat. \ y = y + 0$?
  - Case $y = 0$: $y + 0 = 0 + 0 = 0$
Proving Commutativity of +

\[ \forall x, y \in \text{Nat.} \ x + y = y + x \]

- Case \( x = 0 \). \( \Rightarrow \) \( x + y = 0 + y = y \)
  \( \Rightarrow \) \( \forall y \in \text{Nat.} \ y = y + 0 \) ?
  - 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 +

\[ \forall x, y \in \text{Nat. } x + y = y + x \]

- Case \( x = 0 \). \( \Rightarrow x + y = 0 + y = y \)
  \[\Rightarrow \forall y \in \text{Nat. } y = y + 0 ?\]
  - Case \( y = 0 \): \( y + 0 = 0 + 0 = 0 \)
  - Case \( y = \text{s}(y') \):
    - Induction hypothesis: \( y' = y' + 0 \)
    - \( y + 0 = \text{s}(y') + 0 = \text{s}(y' + 0) = \text{s}(y') = y \)

- Case \( x = \text{s}(x') \). Induction Hypothesis: \( \forall z \in \text{Nat. } x' + z = z + x' \)
  \[ \Rightarrow \forall y \in \text{Nat. } \text{s}(x') + y = y + \text{s}(x') ? \]
Proving Commutativity of $+$

$$\forall x, y \in \text{Nat}. \ x + y = y + x$$

- **Case** $x = 0$. $\Rightarrow x + y = 0 + y = y$
  - $\leadsto \forall y \in \text{Nat}. \ y = y + 0$?
    - 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**: $\forall z \in \text{Nat}. \ x' + z = z + x'$
  - $\leadsto \forall y \in \text{Nat}. \ s(x') + y = y + s(x')$?
    - 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** \( x = 0 \). \( \Rightarrow \) \( x + y = 0 + y = y \)
  \[ \Rightarrow \forall y \in Nat. \ y = y + 0 \]
  - 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: \( \forall z \in Nat. \ x' + z = z + x' \)
  \[ \Rightarrow \forall y \in Nat. \ s(x') + y = y + s(x') \]
  - 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.