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.
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 elements 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 \( \text{List}[\ast] \) parametrized by a domain \( \ast \):
  - Constant: \( [] : \text{List}[\ast] \)
  - Left-concatenation: \( \cdot : \ast \times \text{List}[\ast] \to \text{List}[\ast] \)
- Examples:
  - \( 0 \cdot [] = [0] \)
  - \( 2 \cdot (5 \cdot (8 \cdot ([])))) = 2 \cdot 5 \cdot 8 \cdot [] = [2; 5; 8; 5] \)
  - \( (0 \cdot []) \cdot [] = [0] \)
  - \( [] \cdot [] \neq [] \)
  - \( (0 \cdot []) \cdot ([2 \cdot []] \cdot []) = [[0]; [2]] \)

Defining functions over inductively defined sets

Let \( f : \text{Nat} \to D \). 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 \( + : \text{Nat} \times \text{Nat} \to \text{Nat} \)
- Recursive definition
  
  \[
  \begin{align*}
  0 + x &= x \\
  s(x_1) + x_2 &= s(x_1 + x_2)
  \end{align*}
  \]
- Computation
  
  \[
  \begin{align*}
  s(s(0)) + s(0) &= s(s(0) + s(0)) \\
  &= s(s(0 + s(0))) \\
  &= s(s(s(0)))
  \end{align*}
  \]
Recursion: Another Example

- Append function \( @ : \text{List}[\ast] \times \text{List}[\ast] \rightarrow \text{List}[\ast] \)
- Example: \([2; 5; 7]@[1; 5] = [2; 5; 7; 1; 5]\)
- Recursive definition
  \[
  \emptyset @ \ell = \ell \\
  (a \cdot \ell_1)@\ell_2 = a \cdot (\ell_1@\ell_2)
  \]
- Computation:
  \[
  (2 \cdot 5 \cdot 7 \cdot [])@([1; 5]) = 2 \cdot ((5 \cdot 7 \cdot [])@([1; 5])) \\
  = 2 \cdot 5 \cdot ((7 \cdot [])@([1; 5])) \\
  = 2 \cdot 5 \cdot (([]@([1; 5]))) \\
  = 2 \cdot 5 \cdot 1 \cdot 5 \cdot []
  \]

Composition: Functions can call other functions

- Multiplication \( \ast : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \)
- Recursive definition
  \[
  0 \ast x = 0 \\
  s(x_1) \ast x_2 = x_2 + (x_1 \ast x_2)
  \]
- Computation
  \[
  s^2(0) \ast s^3(0) = s^3(0) + (s(0) \ast s^3(0)) \\
  = s^3(0) + (s^3(0) + 0) \\
  = s(s^3(0)) + (s^3(0) + 0) \\
  = s(s^3(0)) + (s^3(0) + 0) \\
  = s(s(s(s(s(s(0)))))) = s^6(0)
  \]

Composition: Yet Another Example

- Factorial function \( \text{fact} : \text{Nat} \rightarrow \text{Nat} \)
- Recursive definition
  \[
  \text{fact}(0) = 0 \\
  \text{fact}(s(x)) = s(x) \ast \text{fact}(x)
  \]
- Computation
  \[
  \text{fact}(s(0))) = s(0) \ast \text{fact}(s(0)) \\
  = s(0) \ast (s(0) \ast \text{fact}(0)) \\
  = s(0) \ast (s(0) \ast s(0)) \\
  = s(0) \ast s(0) + (s(0) \ast s(0)) \\
  = s(0) \ast s(0) + s(0) \ast s(0) + 0 \ast (s(0) \ast s(0)) \\
  = s(0) \ast s(0) + s(0) \ast s(0) + 0 \ast (s(0) \ast s(0)) \\
  = s(0) + s(0) \\
  = s(s(0))
  \]

Composition: Another Example

- Reverse function \( \text{Rev} : \text{List}[\ast] \rightarrow \text{List}[\ast] \)
- Recursive definition:
  \[
  \text{Rev}([],) = [] \\
  \text{Rev}(a \cdot \ell) = \text{Rev}(\ell)@[a]
  \]
- Computation
  \[
  \text{Rev}([2; 5; 1]) = \text{Rev}([5; 1])@[2] \\
  = (\text{Rev}([1])@[5])@[2] \\
  = ((\text{Rev}([])@[1])@[5])@[2] \\
  = ([1; 5])@[2] \\
  = [1; 5; 2]
  \]

Composition: Functions can call other functions

- Multiplication \( \ast : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \)
- Recursive definition
  \[
  0 \ast x = 0 \\
  s(x_1) \ast x_2 = x_2 + (x_1 \ast x_2)
  \]
- Computation
  \[
  s^2(0) \ast s^3(0) = s^3(0) + (s(0) \ast s^3(0)) \\
  = s^3(0) + (s^3(0) + 0) \\
  = s(s^3(0)) + (s^3(0) + 0) \\
  = s(s(s^3(0))) + (s^3(0) + 0) \\
  = s(s(s(s(s(s(0)))))) = s^6(0)
  \]

Composition: Yet Another Example

- Factorial function \( \text{fact} : \text{Nat} \rightarrow \text{Nat} \)
- Recursive definition
  \[
  \text{fact}(0) = 0 \\
  \text{fact}(s(x)) = s(x) \ast \text{fact}(x)
  \]
- Computation
  \[
  \text{fact}(s(0))) = s(0) \ast \text{fact}(s(0)) \\
  = s(0) \ast (s(0) \ast \text{fact}(0)) \\
  = s(0) \ast (s(0) \ast s(0)) \\
  = s(0) \ast s(0) + (s(0) \ast s(0)) \\
  = s(0) \ast s(0) + s(0) \ast s(0) + 0 \ast (s(0) \ast s(0)) \\
  = s(0) \ast s(0) + s(0) \ast s(0) + 0 \ast (s(0) \ast s(0)) \\
  = s(0) + s(0) \\
  = s(s(0))
  \]

Composition: Functions can call other functions

- Multiplication \( \ast : \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \)
- Recursive definition
  \[
  0 \ast x = 0 \\
  s(x_1) \ast x_2 = x_2 + (x_1 \ast x_2)
  \]
- Computation
  \[
  s^2(0) \ast s^3(0) = s^3(0) + (s(0) \ast s^3(0)) \\
  = s^3(0) + (s^3(0) + 0) \\
  = s(s^3(0)) + (s^3(0) + 0) \\
  = s(s(s^3(0))) + (s^3(0) + 0) \\
  = s(s(s(s(s(s(0)))))) = s^6(0)
  \]

Composition: Yet Another Example

- Factorial function \( \text{fact} : \text{Nat} \rightarrow \text{Nat} \)
- Recursive definition
  \[
  \text{fact}(0) = 0 \\
  \text{fact}(s(x)) = s(x) \ast \text{fact}(x)
  \]
- Computation
  \[
  \text{fact}(s(0))) = s(0) \ast \text{fact}(s(0)) \\
  = s(0) \ast (s(0) \ast \text{fact}(0)) \\
  = s(0) \ast (s(0) \ast s(0)) \\
  = s(0) \ast s(0) + (s(0) \ast s(0)) \\
  = s(0) \ast s(0) + s(0) \ast s(0) + 0 \ast (s(0) \ast s(0)) \\
  = s(0) \ast s(0) + s(0) \ast s(0) + 0 \ast (s(0) \ast s(0)) \\
  = s(0) + s(0) \\
  = s(s(0))
  \]
Functions between different domains

- The Length function \( |·| : \text{List}[⋆] \to \text{Nat} \)

\[
|[]| = 0 \\
|a \cdot ℓ| = s(|ℓ|)
\]

- Sum of the elements \( \sum : \text{List}[\text{Nat}] \to \text{Nat} \)

\[
\sum([]) = 0 \\
\sum(n \cdot ℓ) = n + \sum(ℓ)
\]

Proving facts about functions

- Neutral element:

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

- 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:

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

- Idempotence:

\[
\forall ℓ \in \text{List}[⋆]. \ Rev(Rev(ℓ)) = ℓ
\]

- Kind of distributivity:

\[
\forall ℓ_1, ℓ_2 \in \text{List}[⋆]. \ Rev(ℓ_1 @ ℓ_2) = Rev(ℓ_2) @ Rev(ℓ_1)
\]

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 \( α : D^n \times A \to D \), for every \( e \in E \), define \( f(α(x_1, \cdots, x_n, a), e) \) using \( a \) and \( f(x_1, e), \cdots, f(x_n, e) \).

Structural Induction

Let \( c_1, \ldots, c_m \) be the constants, and let \( α_1, \ldots, α_n \) be the constructors.

\[
P(c_1) \\
\vdots \\
P(c_m)
\]

\[
\bigwedge_{i=1}^{K_1} P(x_i) \Rightarrow P(α_1(x_1, \cdots, x_{K_1})) \\
\vdots \\
\bigwedge_{i=1}^{K_n} P(x_i) \Rightarrow P(α_n(x_1, \cdots, x_{K_n}))
\]

\[
\forall x. \ P(x)
\]
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: $x' * s(0) = s(0) * x' = x'$
  - $s(x') * s(0) = s(0) + (x' * s(0)) = s(0) + x' = s(0 + x') = s(x')$
  - $s(0) * s(x') = s(x') + (0 * s(x')) = s(x' + 0) = s(x')$

Proving Commutativity of $+$

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

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