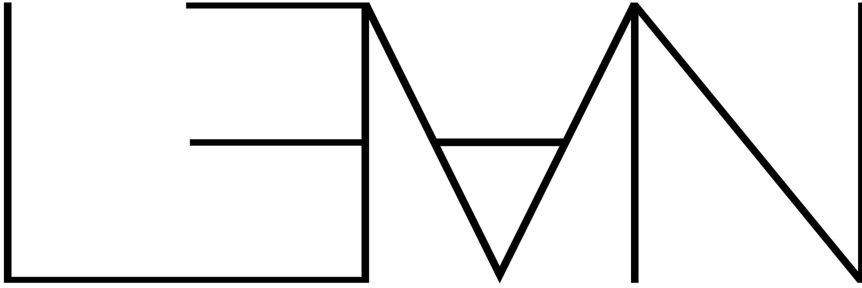


Theorem proving in Lean 3

```
theorem zorn (h : ∀c, chain c → ∃ub, ∀a∈c, a < ub)
  (trans : ∀{a b c}, a < b → b < c → a < c) :
  ∃m, ∀a, m < a → a < m :=
have ∃ub, ∀a∈max_chain, a < ub,
  from h _ $ max_chain_spec.left,
let ⟨ub, (hub : ∀a∈max_chain, a < ub)⟩ := this in
⟨ub, assume a ha,
  have chain (insert a max_chain),
    from chain_insert max_chain_spec.left $ assume b hb _,
      or.inr $ trans (hub b hb) ha,
  have a ∈ max_chain, from
    classical.by_contradiction $ assume h : a ∉ max_chain,
    max_chain_spec.right $ ⟨insert a max_chain, this, ssubset_
  hub a this⟩
```

What is covered in that presentation from Lean documentation

- 1. Introduction – the whole chapter
- 2. Dependent Type Theory – up to 2.7. Namespaces, without 2.8. Dependent Types, 2.9. Implicit Arguments and 2.10. Exercises

Introduction

Terminology

Formal verification

- The use of logic and computational methods to check if the claim, written in a precise mathematical form, is true or false.
- Example of a claim: $(A \wedge B) \Rightarrow (B \wedge A)$
- Verification:
  1. $A \wedge B$
  2. $A \wedge B \Rightarrow A$
  3. $A \wedge B \Rightarrow B$
  4. A
  5. B
  6. $B \Rightarrow (A \Rightarrow B \wedge A)$
  7. $A \Rightarrow B \wedge A$
  8. $B \wedge A$

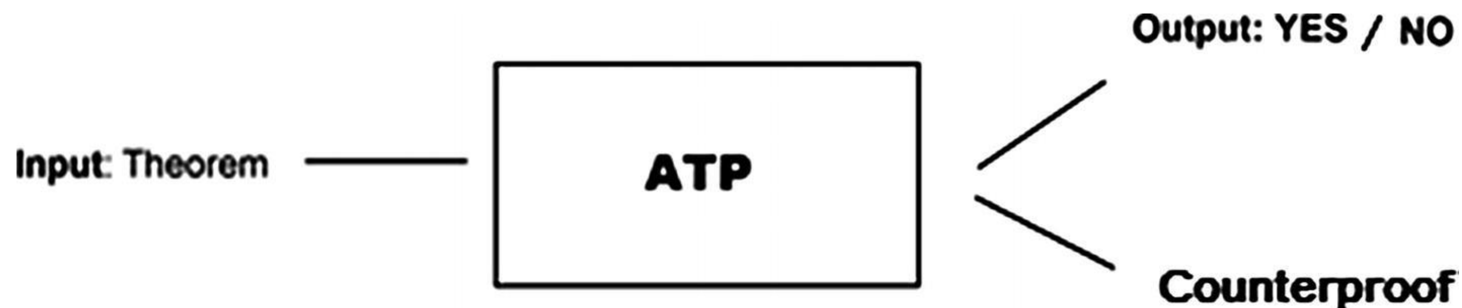
  (Resource of example: Y.Belov, V.Sokolov)

Two ways how we can make a program to provide a proof

- A program can find a proof
- A program can help verify that the given proof is correct

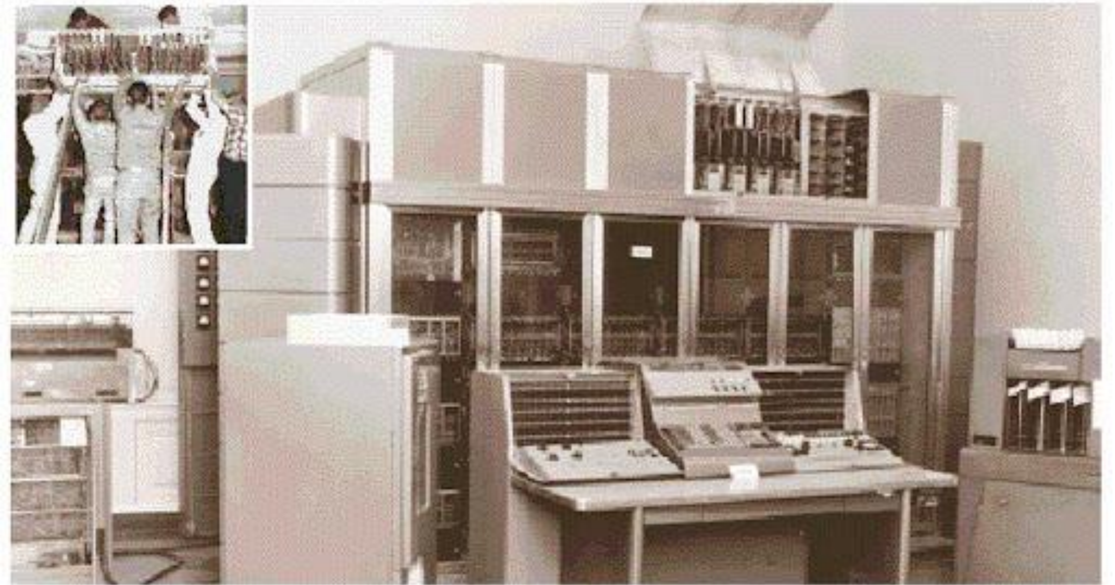
Automated theorem proving

- A theorem proving, where a proof is realized by computer program. The process of proving is based on propositional and first-order logic
- The process is “searching”
  - 👍 powerful, efficient
  - 👎 system can have bugs. Are the results correct?



History of automated theorem proving

- JOHNNIAC (1954, Martin Davis), proved that the sum of two even numbers is even



JOHNNIAC, photo from http://ed-thelen.org

- Logic Theory Machine (1956, Allen Newell, Herbert A. Simon and J. C. Shaw) – ran on JOHNNIAC, constructed proofs from a small set of propositional axioms and three deduction rules: modus ponens, (propositional) variable substitution, and the replacement of formulas by their definition. Managed to prove 38 of the first 52 theorems of the Principia

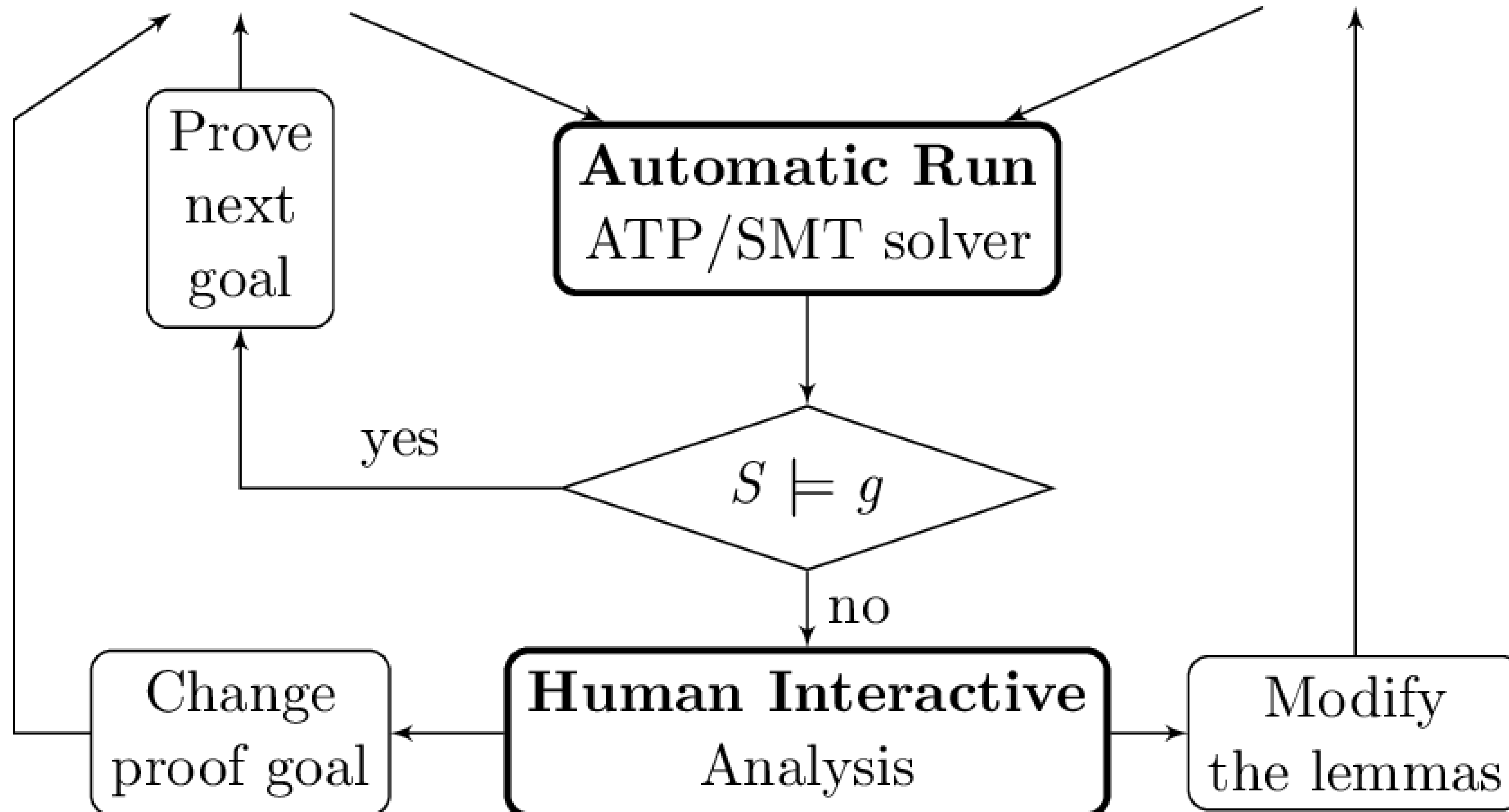
What can we do using ATP

- ✓Check the validity of formulas in propositional and first-order logic:
  - Resolution theorem provers
  - Tableau theorem provers
  - Fast satisfiability solvers
- ✓Search and check the validity expression in specific languages or domains (e.g., of linear or nonlinear expressions over the integers or the real numbers)
- ✓Realize mathematical computations, establish mathematical bounds, find mathematical objects. Calculation equals to proof here.

Interactive theorem prover

- A software, that assists the development of formal proofs
- A human guides the search for proofs. Computer provides and stores the details of proof
- The process is “verifying” – every claim should be supported by a proof in a suitable axiomatic foundation
  - 👎 more input and interaction from user
  - 👍 allow to obtain deeper and more complex proof

Interactive theorem proving



[Resource: Tianxiang Lu on researchgate.net]

The aim of Lean

- To bridge the gap between interactive and automated theorem proving
  - ➢ It has automated tools and methods inside a framework
  - ➢ User interacts with that framework and constructs axiomatic proofs

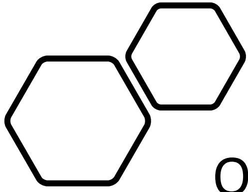
The goal of Lean

“To support both mathematical reasoning and reasoning about complex systems, and to verify claims in both domains”

Programmaticaly speaking, “to prove things about the objects we define”

So, what can we verify in Lean:

- Ordinary mathematical theorems
- Claims about correctness of the programming code, hardware, software, network protocols, mechanical and hybrid systems...



Ordinary mathematical theorems, claims about correctness of programming code, hardware, software, network protocols, mechanical and hybrid systems...

WHY ON THE SAME LEVEL?

Curry–Howard isomorphism

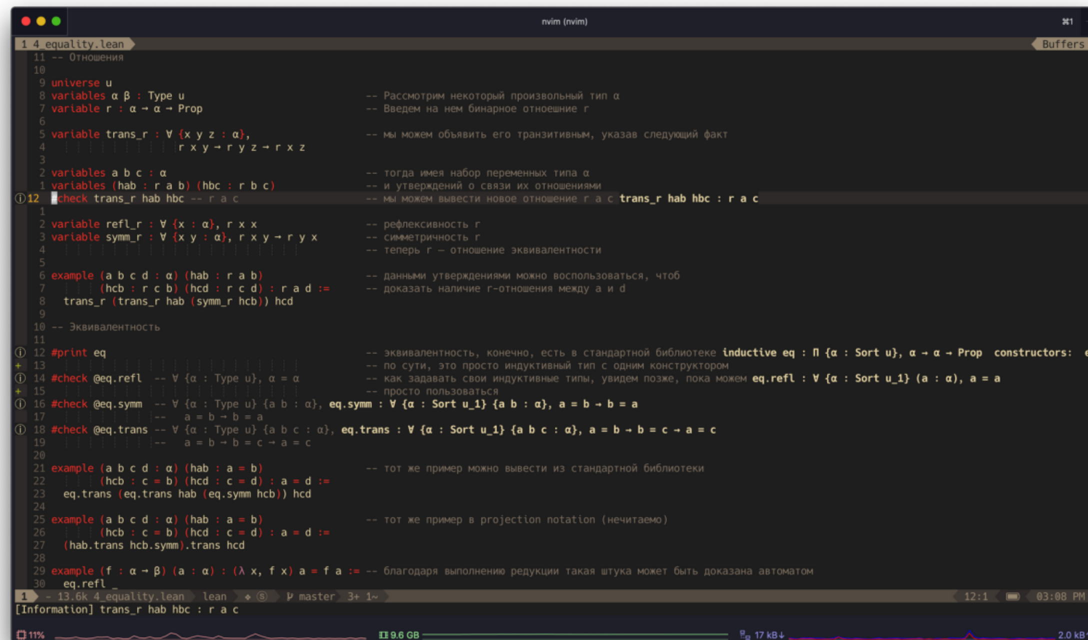
- There is a direct relationship between computer programs and mathematical proofs:
  - ➢ Hardware and software systems are described in mathematical terms, the check of its correctness is a theorem proving
  - ➢ A theorem proof is basically computations, a program. The formula it proves is the type for the program
  - ➢ Proofs can be represented as programs, proofs can be run

How Lean can be used?

- As a programming language, as a system for writing programs with a precise semantics and for reasoning about the functions that the programs compute
- As a metaprogramming language – we can extend the functionality of Lean using the Lean itself

How to use Lean?

- In browser
- Install a program, which is faster and more flexible, than Lean in browser



Lean in VIM editor (download e.g. VS Code if you are a Windows' user)

[Resource: yakovlev.me](yakovlev.me)

2. Dependent Type Theory

Dependent type theory

- Powerful, expressive
- Provides a natural and uniform expressions of complex mathematical assertions, hardware and software specifications
- Every term has a computational behavior (they can be computed)
- Lean is based on a version of dependent type theory *Calculus of Constructions*

Why to use type theory in provers?

- We can track various kinds of mathematical objects: natural numbers, functions on a natural numbers, booleans…
- We can build new types

```
constant m : nat        -- m is a natural number
constant n : nat
constants b1 b2 : bool  -- declare two constants at once

/- check their types -/

#check m              -- output: nat
#check n
#check n + 0          -- nat
#check m * (n + 0)    -- nat
#check b1             -- bool
#check b1 && b2       -- "&&" is boolean and
#check b1 || b2       -- boolean or
#check tt             -- boolean "true"
```

Lean commands

- **constant, constants** – to declare constants, which then available everywhere in code

```
constant a : bool
constants m n : nat
```

- **variable, variables** – to declare constants, that are available inside a block of code (?? ask)
- **section** – to define a block of code, can be nested (sections inside sections). Use it to declare variables for insertion in theorems. It is not necessary to give a name to section, but if you do, it is necessary to close it using the same name

```
section useful
variable x : α
-- more computations here
end useful
```

- **namespace** – for grouping definitions, can be nested (namespaces inside namespaces). Use it to organize data.

```
namespace foo
def a : ℕ := 5
end foo
-- we call it outside like `foo.a`
```

- **def** – to define an object in a form `def foo : α := bar`

```
def double : ℕ → ℕ := λ x, x + x
def double (x : ℕ) : ℕ := x + x
```

- **universe** – to declare a universe variable

```
universe u
constant α : Type u
```

Lean commands

- **#** - the beginning for commands
- **#check** – to check the type of a constant or the type of operation

```
#check (λ x : α, x) a
```

- **#print** – to print something on a screen

```
#print "here I declare constants"
```

- **#reduce** – to evaluate an expression by reducing it to normal form (by carrying out all the computational reductions that are sanctioned by the underlying logic)

```
constant n : nat
#reduce n + 0        -- n
#reduce n + 2        -- nat.succ (nat.succ n)
#reduce 2 + 3        -- 5
```

- **#eval** – the other command to evaluate an expression, less trustworthy, more efficient
- **-- comments, /- also comments -/**

Typing helper

- To type → type `\to`, `\r` or `->`
- To type ℕ type `nat` or `\nat`
- To type × type `\times`
- To type letters like α, β, and γ, type `\a`, `\b`, `\g`

Type

- In type theory, every expression has an associated type – natural number, Boolean, function…
- In Lean everything is a type. Types also have a type of Type, and Type has a type of Type in an infinite hierarchy of types.
- You can define your own types in Lean

Let’s look at types in Lean closer

```
constants m n : nat

constant f : nat → nat           -- type the arrow as "\to" or "\r"
constant f' : nat -> nat         -- alternative ASCII notation
constant f'' : ℕ → ℕ              -- alternative notation for nat
constant p : nat × nat           -- type the product as "\times"
constant q : prod nat nat        -- alternative notation
constant g : nat → nat → nat
constant g' : nat → (nat → nat)  -- has the same type as g!
constant h : nat × nat → nat

constant F : (nat → nat) → nat   -- a "functional"

#check f               -- ℕ → ℕ
#check f n             -- ℕ
#check g m n           -- ℕ
#check g m             -- ℕ → ℕ
#check (m, n)          -- ℕ × ℕ
#check p.1             -- ℕ
#check p.2             -- ℕ
#check (m, n).1        -- ℕ
#check (p.1, n)        -- ℕ × ℕ
#check F f             -- ℕ
```

```
constant m : nat        -- m is a natural number
constant n : nat
constants b1 b2 : bool  -- declare two constants at once

/- check their types -/

#check m              -- output: nat
#check n
#check n + 0          -- nat
#check m * (n + 0)    -- nat
#check b1             -- bool
#check b1 && b2       -- "&&" is boolean and
#check b1 || b2       -- boolean or
#check tt             -- boolean "true"
```

Let’s look at types in Lean closer – 2

```
#check nat                    -- Type
#check bool                   -- Type
#check nat → bool             -- Type
#check nat × bool             -- Type
#check nat → nat              -- ...
#check nat × nat → nat
#check nat → nat → nat
#check nat → (nat → nat)
#check nat → nat → bool
#check (nat → nat) → nat
```

Types have a type of Type

```
constants α β : Type
constant F : Type → Type
constant G : Type → Type → Type

#check α          -- Type
#check F α        -- Type
#check F nat      -- Type
#check G α        -- Type → Type
#check G α β      -- Type
#check G α nat    -- Type
```

We declared new types α and β

```
universe u
constant α : Type u
#check α
```

Declaring a universe type variable

```
constant α : Type _
#check α

constant β : Type*
#check β
```

Declaring a type variable without creating a universe

```
#check Type      -- Type 1
#check Type 1    -- Type 2
#check Type 2    -- Type 3
#check Type 3    -- Type 4
#check Type 4    -- Type 5
```

An infinite hierarchy of types lies in foundation of Lean: Type 0 (or Type) is a universe of "small" or "ordinary" types. Type 1 is a larger universe of types, which contains Type 0 as an element. Type 2 is a larger universe of types, which contains Type 1 as an element, etc.

Functions in Lean

- `nat → nat` denotes the type of functions that take a natural number as input and return a natural number as output

```
constant f : nat → nat
```

- `(m, n)` denotes the ordered pair of `m` and `n`, and if `p` is a pair, `p.1` and `p.2` denote the two projections

```
constants m n : nat
constant p : nat × nat
#check p.1          -- ℕ
#check (m, n)       -- ℕ × ℕ
#check (p.1, n)     -- ℕ × ℕ
```

- The application of a function `f` to a value `x` is denoted `f x`

```
#check f x          -- ℕ
```

- When writing type expressions, arrows associate to the right; for example, in code `constant g : nat → nat → nat`, the type of `g` is `nat → (nat → nat)`
  - ➢ `g` is a function that takes natural numbers and returns another function that takes a natural number and returns a natural number – at the end, the number is returned
  - ➢ This process is called ***currying***

Creating a function from another expression

- We need a lambda abstraction process here (we need to turn terms and parts of terms into a variable)
- We temporarily postulate a variable $x : \alpha$, and then we can construct an expression $t : \beta$
- The expression fun $x : \alpha, t$, or, equivalently, $\lambda\, x : \alpha, t$, is an object of type $\alpha \to \beta$
- It is a function from α to β which maps any value x to the value t, which depends on x
- "Let f be the function which maps any natural number x to x + 5"

```
#check fun x : nat, x + 5
#check λ x : nat, x + 5
```

Some terminology…

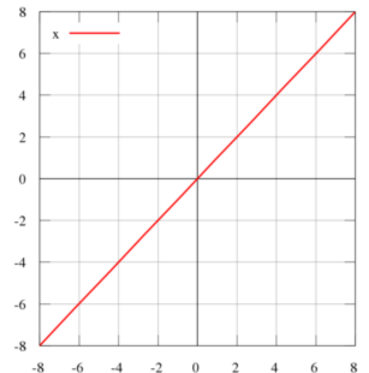
- Bound variable – a placeholder with the given scope

  λ x : α, t - x is a placeholder, whose "scope" doesn't extend beyond t

- Alpha equivalent expressions – the expressions that are the same up to a renaming of bound variables, they are considered “the same.”

  λ (b : β) (x : α), x is equal to λ (u : β) (z : α), z

- Identity function – a function that always returns the same value that was used as its argument



Graph of the identity function on the real numbers, Resource: Wikipedia

Some terminology – 2

- Constant function – a function whose (output) value is the same for every input value, e.g., $f(x) = 4$ is a constant function
- Function composition – applying one function to the results of another

```
constants α β   : Type
constants a1 a2 : α
constants b1 b2 : β

constant f : α → α
constant g : α → β
constant h : α → β → α
constant p : α → α → bool

#check fun x : α, f x                      -- α → α
#check λ x : α, f x                        -- α → α
#check λ x : α, f (f x)                    -- α → α
#check λ x : α, h x b1                     -- α → α
#check λ y : β, h a1 y                     -- β → α
#check λ x : α, p (f (f x)) (h (f a1) b2)  -- α → bool
#check λ x : α, λ y : β, h (f x) y         -- α → β → α
#check λ (x : α) (y : β), h (f x) y        -- α → β → α
#check λ x y, h (f x) y                    -- α → β → α
```

Lean interprets the final three examples as the same expression; in the last expression, Lean infers the type of x and y from the types of f and h

```
constants α β γ : Type
constant f : α → β
constant g : β → γ
constant b : β

#check λ x : α, x          -- α → α
#check λ x : α, b          -- α → β
#check λ x : α, g (f x)    -- α → γ
#check λ x, g (f x)
```

The expression λ x : α, x denotes the identity function on α, the expression λ x : α, b denotes the constant function that always returns b, and λ x : α, g (f x), denotes the composition of f and g

```
#check λ b : β, λ x : α, x     -- β → α → α
#check λ (b : β) (x : α), x    -- β → α → α
#check λ (g : β → γ) (f : α → β) (x : α), g (f x)
                              -- (β → γ) → (α → β) → α → γ
```

We can abstract over any of the constants in the previous definitions

```
#check λ (α β : Type*) (b : β) (x : α), x
#check λ (α β γ : Type*) (g : β → γ) (f : α → β) (x : α), g (f x)
```

We can even abstract over the type

The function that takes three types, α, β, and γ, and two functions, g : β → γ and f : α → β, and returns the composition of g and f

#reduce

- Beta reduction - the process of simplifying an expression. E.g., simplifying (λ x, t)s to t[s/x] - t with s substituted for the variable x
- Beta equivalent terms - terms that beta reduce to a common term
- Definitionally equal term – two terms that reduce to the same value
- In dependent type theory every term has a computational behavior, and supports a notion of reduction, or normalization

```
constants α β γ : Type
constant f : α → β
constant g : β → γ
constant h : α → α
constants (a : α) (b : β)

#reduce (λ x : α, x) a                    -- a
#reduce (λ x : α, b) a                    -- b
#reduce (λ x : α, b) (h a)                -- b
#reduce (λ x : α, g (f x)) a              -- g (f a)

#reduce (λ (v : β → γ) (u : α → β) x, v (u x)) g f a   -- g (f a)

#reduce (λ (Q R S : Type*) (v : R → S) (u : Q → R) (x : Q),
       v (u x)) α β γ g f a          -- g (f a)
```

```
constants m n : nat
constant b : bool

#print "reducing pairs"
#reduce (m, n).1          -- m
#reduce (m, n).2          -- n

#print "reducing boolean expressions"
#reduce tt && ff          -- ff
#reduce ff && b           -- ff
#reduce b && ff           -- bool.rec ff ff b

#print "reducing arithmetic expressions"
#reduce n + 0             -- n
#reduce n + 2             -- nat.succ (nat.succ n)
#reduce 2 + 3             -- 5
```

How we are working in Lean most of the time?

- We define objects and prove things about them

```
def double : ℕ → ℕ := λ x, x + x
def square : ℕ → ℕ := λ x, x * x
def do_twice : (ℕ → ℕ) → ℕ → ℕ := λ f x, f (f x)
```

```
def double (x : ℕ) : ℕ := x + x
def square (x : ℕ) := x * x
def do_twice (f : ℕ → ℕ) (x : ℕ) : ℕ := f (f x)
```

```
def do_twice (α : Type*) (h : α → α) (x : α) : α := h (h x)
def do_twice (h : α → α) (x : α) : α := h (h x)
```

```
variables (α β γ : Type*)
variables (g : β → γ) (f : α → β) (h : α → α)
variable x : α

def do_twice := h (h x)
```

Local definitions using let

- let a := t1 in t2 means – all a in t2 will be replaced by t1
- #reduce let y := 2 + 2, z := y + y in z * z   -- 64 - all z are replaced by y + y, which are 4+4, so z * z equals to 64

```
def t (x : ℕ) : ℕ :=
let y := x + x in y * y
#reduce t 2   -- 16
```

Namespaces

- Like sections, nested namespaces have to be closed in the order they are opened
- Namespaces cannot be declared inside a section

```
namespace foo
def a : ℕ := 5
#print "inside foo"
#check a
end foo
#print "outside the namespace"
-- #check a  -- error
open foo
#print "opened foo"
#check a
```

Resources

- https://en.wikipedia.org/wiki/Curry%E2%80%93Howard_correspondence
- https://yakovlev.me/lean-intro/
- https://leanprover.github.io/theorem_proving_in_lean/
- https://commons.wikimedia.org/w/index.php?curid=66273005
- https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B4%D0%BE%D0%BA%D0%B0%D0%B7%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE#cite_note-1