# Type Systems and Functional Programming

S.l. dr. ing. Mihnea Muraru
mmihnea@gmail.com
Computer Science Department

Fall 2016



# Part I

# Introduction



## Contents

1. Objectives
2. Functional programming



## Contents

1. Objectives
2. Functional programming



## Grading

- Lab: 60, $\geq 30$
- Exam: 40, $\geq 20$
- Final grade $\geq 50$



## Course objectives

- Studying the particularities of functional programming, such as *lazy evaluation* and *type systems* of different strengths
- Learning advanced mechanisms of the Haskell language, which are impossible or difficult to simulate in other languages
- Applying this apparatus to modeling practical problems, e.g. program synthesis, lazy search, probability spaces, genetic algorithms . . .



## One of the lab outcomes

An evaluator for a functional language, equipped with a type synthesizer



## Contents

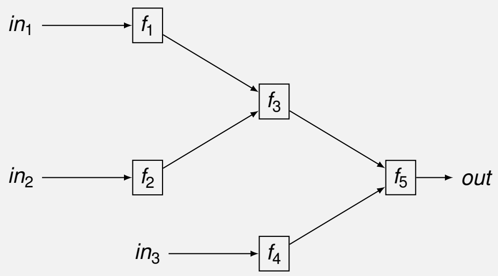
1. Objectives
2. Functional programming

## Functional programming features

- Mathematical functions, as value transformers
- Functions as first-class values
- No side effects or state
- Immutability
- Referential transparency
- Lazy evaluation
- Recursion
- Higher-order functions



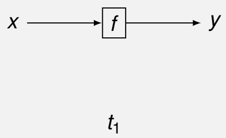
## Functional flow






## Stateless computation

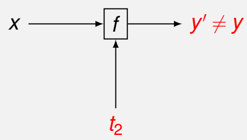
Output dependent on input exlcusively:






## Stateful computation

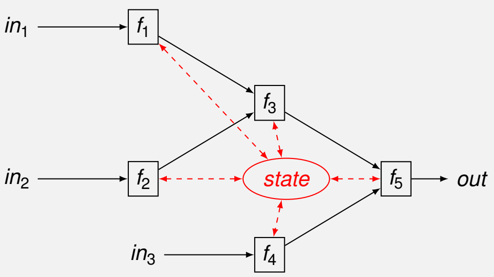
Output dependent on input and time:






## Functional flow

Pure






## Functional programming features

- Mathematical functions, as value transformers
- Functions as first-class values
- No side effects or state
- Immutability
- Referential transparency
- Lazy evaluation
- Recursion
- Higher-order functions



## Why functional programming?

- Simple processing model; equational reasoning
- Declarative
- Modularity, composability, reuse (lazy evaluation as glue)
- Exploration of huge or formally infinite search spaces
- Embedded Domain Specific Languages (EDSLs)
- Massive parallelization
- Type systems and logic, inextricably linked
- Automatic program verification and synthesis



# Part II

# Untyped Lambda Calculus

## Contents


3 Introduction
4 Lambda expressions
5 Reduction
6 Normal forms
7 Evaluation order




## Contents


3 Introduction
4 Lambda expressions
5 Reduction
6 Normal forms
7 Evaluation order




## Untyped lambda calculus

- Model of computation — Alonzo Church, 1932
- Equivalent to the Turing machine (see the Church-Turing thesis)
- Main building block: the function
- Computation: evaluation of function applications, through textual substitution
- Evaluate = obtain a value (a *function*)!
- No side effects or state



## Applications

- Theoretical basis of numerous languages:
  - LISP
  - Scheme
  - Haskell
  - ML
  - F#
  - Clean
  - Clojure
  - Scala
  - Erlang
- Formal program verification, due to its simple execution model



## Contents


3 Introduction
4 Lambda expressions
5 Reduction
6 Normal forms
7 Evaluation order




## $\lambda$-expressions
Definition

**Definition 4.1 ($\lambda$-expression).**

- Variable: a variable $x$ is a $\lambda$-expression
- Function: if $x$ is a variable and $E$ is a $\lambda$-expression, then $\lambda x.E$ is a $\lambda$-expression, which stands for an anonymous, unary function, with the formal argument $x$ and the body $E$
- Application: if $E$ and $A$ are $\lambda$-expressions, then $(E\ \ A)$ is a $\lambda$-expression, which stands for the application of the expression $E$ onto the actual argument $A$.



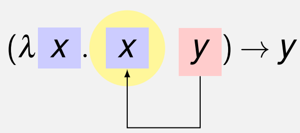
## $\lambda$-expressions
Examples

**Example 4.2 ($\lambda$-expressions).**

- $x \rightarrow$ variable $x$
- $\lambda x.x$ : the identity function
- $\lambda x.\lambda y.x$ : a function with another function as body!
- $(\lambda x.x\ \ y)$ : the application of the identity function onto the actual argument $y$
- $(\lambda x.(x\ \ x)\ \ \lambda x.x)$



## Intuition on application evaluation



## Variable occurrences
### Definitions

**Definition 4.3 (Bound occurrence).**
An occurrence $x_n$ of a variable $x$ is bound in the expression $E$ iff:
- $E = \lambda x.F$ or
- $E = \dots\ \lambda x_n.F\ \dots$ or
- $E = \dots\ \lambda x.F\ \dots$ and $x_n$ appears in $F$.

**Definition 4.4 (Free occurrence).**
A variable occurrence is free in an expression iff it is not bound in that expression.

Bound/ free occurrence w.r.t. a given expression!



## Variable occurrences
### Examples

**Example 4.5 (Bound and free variables).**
In the expression $E = (\lambda x.x\ \ x)$, we emphasize the occurrences of $x$:

$$E = (\lambda x_1.\underbrace{x_2}_{F}\ \ x_3).$$

- $x_1$, $x_2$ bound in $E$
- $x_3$ free in $E$
- $x_2$ free in $F$!
- $x$ free in $E$ and $F$



## Variable occurrences
### Examples

**Example 4.6 (Bound and free variables).**
In the expression $E = (\lambda x.\lambda z.(z\ \ x)\ \ (z\ \ y))$, we emphasize the occurrences of $x$, $y$, $z$:

$$E = (\lambda x_1.\overbrace{\lambda z_1.(z_2\ \ x_2)}^{F}\ \ (z_3\ \ y_1)).$$

- $x_1$, $x_2$, $z_1$, $z_2$ bound in $E$
- $y_1$, $z_3$ free in $E$
- $z_1$, $z_2$ bound in $F$
- $x_2$ free in $F$
- $x$ bound in $E$, but free in $F$
- $y$ free in $E$
- $z$ free in $E$, but bound in $F$



## Variables
### Definitions

**Definition 4.7 (Bound variable).**
A variable is bound in an expression iff all its occurrences are bound in that expression.

**Definition 4.8 (Free variable).**
A variable is free in an expression iff it is not bound in that expression i.e., iff at least one of its occurrences is free in that expression.

Bound/ free variable w.r.t. a given expression!



## Variable occurrences
### Examples

**Example 4.5 (Bound and free variables).**
In the expression $E = (\lambda x.x\ \ x)$, we emphasize the occurrences of $x$:

$$E = (\lambda x_1.\underbrace{x_2}_{F}\ \ x_3).$$

- $x_1$, $x_2$ bound in $E$
- $x_3$ free in $E$
- $x_2$ free in $F$!
- $x$ free in $E$ and $F$



## Variable occurrences
### Examples

**Example 4.6 (Bound and free variables).**
In the expression $E = (\lambda x.\lambda z.(z\ \ x)\ \ (z\ \ y))$, we emphasize the occurrences of $x$, $y$, $z$:

$$E = (\lambda x_1.\overbrace{\lambda z_1.(z_2\ \ x_2)}^{F}\ \ (z_3\ \ y_1)).$$

- $x_1$, $x_2$, $z_1$, $z_2$ bound in $E$
- $y_1$, $z_3$ free in $E$
- $z_1$, $z_2$ bound in $F$
- $x_2$ free in $F$
- $x$ bound in $E$, but free in $F$
- $y$ free in $E$
- $z$ free in $E$, but bound in $F$



## Free and bound variables

**Free variables**
- $FV(x) = \{x\}$
- $FV(\lambda x.E) = FV(E) \setminus \{x\}$
- $FV((E_1\ \ E_2)) = FV(E_1) \cup FV(E_2)$

**Bound variables**
- $BV(x) = \emptyset$
- $BV(\lambda x.E) = BV(E) \cup \{x\}$
- $BV((E_1\ \ E_2)) = BV(E_1) \setminus FV(E_2) \cup BV(E_2) \setminus FV(E_1)$



## Closed expressions

**Definition 4.9 (Closed expression).**
An expression that does not contain any free variables.

**Example 4.10 (Closed expressions).**
- $(\lambda x.x\ \ \lambda x.\lambda y.x)$ : closed
- $(\lambda x.x\ \ a)$ : open, since $a$ is free

Remarks:
- Free variables may stand for other $\lambda$-expressions, as in $\lambda x.((+\ \ x)\ \ 1)$.
- Before evaluation, an expression must be brought to the closed form.
- The substitution process must terminate.

## Contents


3 Introduction

4 Lambda expressions

5 Reduction

6 Normal forms

7 Evaluation order




## $\beta$-reduction

### Definitions

**Definition 5.1 ($\beta$-reduction).**
The evaluation of the application $(\lambda x.E\ A)$, by substituting every free occurrence of the formal argument, $x$, in the function body, $E$, with the actual argument, $A$:
$(\lambda x.E\ A) \rightarrow_\beta E_{[A/x]}$.

**Definition 5.2 ($\beta$-redex).**
The application $(\lambda x.E\ A)$.



## $\beta$-reduction

### Examples

**Example 5.3 ($\beta$-reduction).**

- $(\lambda x.x\ y) \rightarrow_\beta x_{[y/x]} \rightarrow y$
- $(\lambda x.\lambda x.x\ y) \rightarrow_\beta \lambda x.x_{[y/x]} \rightarrow \lambda x.x$
- $(\lambda x.\lambda y.x\ y) \rightarrow_\beta \lambda y.x_{[y/x]} \rightarrow \lambda y.y$

  Wrong! The free variable $y$ becomes bound, changing its meaning!



## $\beta$-reduction

### Collisions

- Problem: within the expression $(\lambda x.E\ A)$:
  - $FV(A) \cap BV(E) = \emptyset \Rightarrow$ correct reduction always
  - $FV(A) \cap BV(E) \neq \emptyset \Rightarrow$ potentially wrong reduction
- Solution: rename the bound variables in $E$, that are free in $A$

**Example 5.4 (Bound variable renaming).**
$(\lambda x.\lambda y.x\ y) \rightarrow (\lambda x.\lambda z.x\ y) \rightarrow_\beta \lambda z.x_{[y/x]} \rightarrow \lambda z.y$



## $\alpha$-conversion

### Definition

**Definition 5.5 ($\alpha$-conversion).**
Systematic relabeling of bound variables in a function: $\lambda x.E \rightarrow_\alpha \lambda y.E_{[y/x]}$. Two conditions must be met.

**Example 5.6 ($\alpha$-conversion).**

- $\lambda x.y \rightarrow_\alpha \lambda y.y_{[y/x]} \rightarrow \lambda y.y$ : Wrong!
- $\lambda x.\lambda y.x \rightarrow_\alpha \lambda y.\lambda y.x_{[y/x]} \rightarrow \lambda y.\lambda y.y$ : Wrong!

Conditions:

- $y$ is not free in $E$
- a free occurrence in $E$ stays free in $E_{[y/x]}$



## $\alpha$-conversion

### Examples

**Example 5.7 ($\alpha$-conversion).**

- $\lambda x.(x\ y) \rightarrow_\alpha \lambda z.(z\ y)$ : Correct!
- $\lambda x.\lambda x.(x\ y) \rightarrow_\alpha \lambda y.\lambda x.(x\ y)$ : Wrong!
  $y$ is free in $\lambda x.(x\ y)$.
- $\lambda x.\lambda y.(y\ x) \rightarrow_\alpha \lambda y.\lambda y.(y\ y)$ : Wrong!
  The free occurrence of $x$ in $\lambda y.(y\ x)$ becomes bound, after substitution, in $\lambda y.(y\ y)$.
- $\lambda x.\lambda y.(y\ y) \rightarrow_\alpha \lambda y.\lambda y.(y\ y)$ : Correct!



## Reduction

### Definitions

**Definition 5.8 (Reduction step).**
A sequence made of a possible $\alpha$-conversion, followed by a $\beta$-reduction, such that the second produces no collisions: $E_1 \rightarrow E_2 \equiv E_1 \rightarrow_\alpha E_3 \rightarrow_\beta E_2$.

**Definition 5.9 (Reduction sequence).**
A string of zero or more reduction steps: $E_1 \rightarrow^* E_2$. It is an element of the reflexive transitive closure of relation $\rightarrow$.



## Reduction

### Examples

**Example 5.10 (Reduction).**

- $((\lambda x.\lambda y.(y\ x)\ y)\ \lambda x.x)$
  $\rightarrow (\lambda z.(z\ y)\ \lambda x.x)$
  $\rightarrow (\lambda x.x\ y)$
  $\rightarrow y$
- $((\lambda x.\lambda y.(y\ x)\ y)\ \lambda x.x) \rightarrow^* y$

## Reduction
Properties

- Reduction step ⇒ reduction sequence:

$$E_1 \to E_2 \Rightarrow E_1 \to^* E_2$$

- Reflexivity:

$$E \to^* E$$

- Transitivity:

$$E_1 \to^* E_2 \wedge E_2 \to^* E_3 \Rightarrow E_1 \to^* E_3$$



## Contents


3 Introduction
4 Lambda expressions
5 Reduction
6 Normal forms
7 Evaluation order




## Questions

1. When does the computation terminate? Does it always?
   - NO
2. Does the answer depend on the reduction sequence?
   - YES
3. If the computation terminates for distinct reduction sequences, do we always get the same result?
   - YES
4. If the result is unique, how do we safely obtain it?
   - Left-to-right reduction



## Normal forms

**Definition 6.1 (Normal form).**
The form of an expression that cannot be reduced i.e., that contains no $\beta$-redexes.

**Definition 6.2 (Functional normal form, FNF).**
$\lambda x.E$, even if $E$ contains $\beta$-redexes.

**Example 6.3 (Normal forms).**
$(\lambda x.\lambda y.(x\ y)\ \lambda x.x) \to_{FNF} \lambda y.(\lambda x.x\ y) \to_{NF} \lambda y.y$

FNF is used in programming, where the function body is evaluated only when the function is effectively applied.



## Reduction termination (reducibility)

**Example 6.4.**
$\Omega \equiv (\lambda x.(x\ x)\ \lambda x.(x\ x)) \to (\lambda x.(x\ x)\ \lambda x.(x\ x)) \to^* \ldots$
$\Omega$ does not have a terminating reduction sequence.

**Definition 6.5 (Reducible expression).**
An expression that has a terminating reduction sequence.

$\Omega$ is irreducible.



## Questions

1. When does the computation terminate? Does it always?
   - NO
2. Does the answer depend on the reduction sequence?
   - YES
3. If the computation terminates for distinct reduction sequences, do we always get the same result?
   - YES
4. If the result is unique, how do we safely obtain it?
   - Left-to-right reduction



## Reduction sequences

**Example 6.6 (Reduction sequences).**

$$E = (\lambda x.y\ \Omega)$$

- $\xrightarrow{1} y$
- $\xrightarrow{2} E \xrightarrow{1} y$
- $\xrightarrow{2} E \xrightarrow{2} E \xrightarrow{1} y$
- $\ldots$
- $\xrightarrow{2^n 1}{}^* y,\ n \geq 0$
- $\xrightarrow{2^\infty}{}^* \ldots$

- $E$ has a nonterminating reduction sequence, but still has a normal form, $y$. $E$ is reducible, $\Omega$ is not.
- The length of terminating reduction sequences is unbounded.

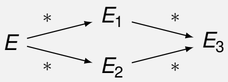


## Questions

1. When does the computation terminate? Does it always?
   - NO
2. Does the answer depend on the reduction sequence?
   - YES
3. If the computation terminates for distinct reduction sequences, do we always get the same result?
   - YES
4. If the result is unique, how do we safely obtain it?
   - Left-to-right reduction

# Normal form uniqueness
## Results

**Theorem 6.7 (Church-Rosser / diamond).**
*If $E \rightarrow^* E_1$ and $E \rightarrow^* E_2$, then there is an $E_3$ such that $E_1 \rightarrow^* E_3$ and $E_2 \rightarrow^* E_3$.*




**Corollary 6.8 (Normal form uniqueness).**
*If an expression is reducible, its normal form is unique. It corresponds to the value of that expression.*



# Normal form uniqueness
## Examples

**Example 6.9 (Normal form uniqueness).**

$$(\lambda x.\lambda y.(x\ y)\ (\lambda x.x\ y))$$

- $\rightarrow \lambda z.((\lambda x.x\ y)\ z) \rightarrow \lambda z.(y\ z) \rightarrow_\alpha \lambda a.(y\ a)$
- $\rightarrow (\lambda x.\lambda y.(x\ y)\ y) \rightarrow \lambda w.(y\ w) \rightarrow_\alpha \lambda a.(y\ a)$

- Normal form: class of expressions, equivalent under systematic relabeling
- Value: distinguished member of this class



# Structural equivalence

**Definition 6.10 (Structural equivalence).**
Two expressions are structurally equivalent iff they both reduce to the same expression.

**Example 6.11 (Structural equivalence).**
$\lambda z.((\lambda x.x\ y)\ z)$ and $(\lambda x.\lambda y.(x\ y)\ y)$ in Example 6.9.



# Computational equivalence

**Definition 6.12 (Computational equivalence).**
Two expressions are computationally equivalent iff they the behave in the same way when applied onto the same arguments.

**Example 6.13 (Computational equivalence).**

$$E_1 = \lambda y.\lambda x.(y\ x)$$
$$E_2 = \lambda x.x$$

- $((E_1\ a)\ b) \rightarrow^* (a\ b)$
- $((E_2\ a)\ b) \rightarrow^* (a\ b)$
- $E_1 \not\rightarrow^* E_2$ and $E_2 \not\rightarrow^* E_1$ (not structurally equivalent)



# Questions

1. When does the computation terminate?
   Does it always?
   - NO
2. Does the answer depend on the reduction sequence?
   - YES
3. If the computation terminates for distinct reduction sequences, do we always get the same result?
   - YES
4. If the result is unique, how do we safely obtain it?
   - Left-to-right reduction



# Reduction order
## Definitions and examples

**Definition 6.14 (Left-to-right reduction step).**
The reduction of the outermost leftmost $\beta$-redex.

**Example 6.15 (Left-to-right reduction).**
$(\underline{(\lambda x.x\ \lambda x.y)}\ (\lambda x.(x\ x)\ \lambda x.(x\ x))) \rightarrow \underline{(\lambda x.y\ \Omega)} \rightarrow y$

**Definition 6.16 (Right-to-left reduction step).**
The reduction of the innermost rightmost $\beta$-redex.

**Example 6.17 (Right-to-left reduction).**
$((\lambda x.x\ \lambda x.y)\ \underline{(\lambda x.(x\ x)\ \lambda x.(x\ x))}) \rightarrow (\lambda x.y\ \underline{\Omega}) \rightarrow \ldots$



# Reduction order
## Which one is better?

**Theorem 6.18 (Normalization).**
*If an expression is reducible, its left-to-right reduction terminates.*

The theorem does not guarantee the termination for any expression, but only for reducible ones!



# Questions

1. When does the computation terminate?
   Does it always?
   - NO
2. Does the answer depend on the reduction sequence?
   - YES
3. If the computation terminates for distinct reduction sequences, do we always get the same result?
   - YES
4. If the result is unique, how do we safely obtain it?
   - Left-to-right reduction

# Contents


3 Introduction

4 Lambda expressions

5 Reduction

6 Normal forms

7 Evaluation order




# Evaluation order

**Definition 7.1 (Applicative-order evaluation).**
Corresponds to right-to-left reduction. Function arguments are evaluated before the function is applied.

**Definition 7.2 (Strict function).**
A function that uses applicative-order evaluation.

**Definition 7.3 (Normal-order evaluation).**
Corresponds to left-to-right reduction. Function arguments are evaluated when needed.

**Definition 7.4 (Non-strict function).**
A function that uses normal-order evaluation.



# In practice I

Applicative-order evaluation employed in most programming languages, due to efficiency — one-time evaluation of arguments: C, Java, Scheme, PHP, etc.

**Example 7.5 (Applicative-order evaluation in Scheme).**

```
((λ (x) (+ x x)) (+ 2 3))
→ ((λ (x) (+ x x)) 5)
→ (+ 5 5)
→ 10
```



# In practice II

Lazy evaluation (a kind of normal-order evaluation) in Haskell: on-demand evaluation of arguments, allowing for interesting constructions

**Example 7.6 (Lazy evaluation in Haskell).**

```
((\x -> x + x) (2 + 3))
→ (2 + 3) + (2 + 3)
→ 5 + 5
→ 10
```

Need for non-strict functions, even in applicative languages: `if`, `and`, `or`, etc.



# Summary

- Lambda calculus: model of computation, underpinned by functions and textual substitution
- Bound/free variables and variable occurrences w.r.t. an expression
- $\beta$-reduction, $\alpha$-conversion, reduction step, reduction sequence, reduction order, normal forms
- Left-to-right reduction (normal-order evaluation): always terminates for reducible expressions
- Right-to-left reduction (applicative-order evaluation): more efficient but no guarantee on termination even for reducible expressions!



# Part III

# Lambda Calculus as a Programming Language



# Contents


8 The $\lambda_0$ language

9 Abstract data types (ADTs)

10 Implementation

11 Recursion

12 Language specification




# Contents


8 The $\lambda_0$ language

9 Abstract data types (ADTs)

10 Implementation

11 Recursion

12 Language specification

## Purpose

- Proving the expressive power of lambda calculus
- Hypothetical $\lambda$-machine
- Machine code: $\lambda$-expressions — the $\lambda_0$ language
- Instead of
  - bits
  - bit operations,

  we have
  - structured strings of symbols
  - reduction — textual substitution



## $\lambda_0$ features

- Instructions:
  - $\lambda$-expressions
  - top-level variable bindings: *variable* $\equiv_{\text{def}}$ *expression*
    e.g., *true* $\equiv_{\text{def}} \lambda x.\lambda y.x$
- Values represented as functions
- Expressions brought to the closed form, prior to evaluation
- Normal-order evaluation
- Functional normal form (see Definition 6.2)
- No predefined types!



## Shorthands

- $\lambda x_1.\lambda x_2.\lambda \ldots .\lambda x_n.E \rightarrow \lambda x_1 x_2 \ldots x_n.E$
- $((\ldots((E\ A_1)\ A_2)\ \ldots)\ A_n) \rightarrow (E\ A_1\ A_2\ \ldots\ A_n)$



## Purpose of types

- Way of expressing the programmer's intent
- Documentation: which operators act onto which objects
- Particular representation for values of different types: `1`, `"Hello"`, `#t`, etc.
- Optimization of specific operations
- Error prevention
- Formal verification



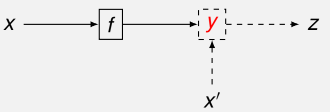
## No types

### How are objects represented?

- A number, list or tree potentially designated by the same value e.g.,

  number 3 $\rightarrow \lambda x.\lambda y.x \leftarrow$ list (() () ())
- Both values and operators represented by functions — context-dependent meaning

  number 3 $\rightarrow \lambda x.\lambda y.x \leftarrow$ operator *car*
- Value applicable onto another value, as an operator!






## No types

### How is correctness affected?

- Inability of the $\lambda$ machine to
  - interpret the meaning of expressions
  - ensure their correctness
- Every operator applicable onto every value
- Both aspects above delegated to the programmer
- Erroneus constructs accepted without warning, but computation ended with
  - values with no meaning or
  - expressions that are neither values, nor reducible
    e.g., $(x\ x)$



## No types

### Consequences

- Enhanced representational flexibility
- Useful when the uniform representation of objects, as lists de symbols, is convenient
- Increased error-proneness
- Program instability
- Difficulty of verification and maintenance



## So...

- How do we employ the $\lambda_0$ language in everyday programming?
- How do we represent usual values — numbers, booleans, lists, etc. — and their corresponding operators?

## Contents


8 The $\lambda_0$ language

9 Abstract data types (ADTs)

10 Implementation

11 Recursion

12 Language specification




## Definition

**Definition 9.1 (Abstract data type, ADT).**
Mathematical model of a set of values and their corresponding operations.

**Example 9.2 (ADTs).**
*Natural*, *Bool*, *List*, *Set*, *Stack*, *Tree*, ... $\lambda$-expression!

Components:

- base constructors: how are values built
- operators: what can be done with these values
- axioms: how



## The *Natural* ADT

Base constructors and operators

- Base constructors:
  - $zero : \to Natural$
  - $succ : Natural \to Natural$
- Operators:
  - $zero? : Natural \to Bool$
  - $pred : Natural \setminus \{zero\} \to Natural$
  - $add : Natural^2 \to Natural$



## The *Natural* ADT

Axioms

- *zero?*
  - $(zero?\ zero) = T$
  - $(zero?\ (succ\ n)) = F$
- *pred*
  - $(pred\ (succ\ n)) = n$
- *add*
  - $(add\ zero\ n) = n$
  - $(add\ (succ\ m)\ n) = (succ\ (add\ m\ n))$



## Providing axioms

- One axiom for each (operator, base constructor) pair
- More — useless
- Less — insufficient for completely specifying the operators



## From ADTs to functional programming

Exemple

- Axiome:
  - $add(zero, n) = n$
  - $add(succ(m), n) = succ(add(m, n))$
- Scheme:

```
(define add
  (lambda (m n)
    (if (zero? m) n
        (+ 1 (add (- m 1) n)))))
```

- Haskell:

```
add 0 n = n
add (m + 1) n = 1 + (add m n)
```



## From ADTs to functional programming

Discussion

- Proving ADT correctness — structural induction
- Proving properties of $\lambda$-expressions, seen as values of an ADT with 3 base constructors!
- Functional programming — reflection of mathematical specifications
- Recursion — natural instrument, inherited from axioms
- Applying formal methods on the recursive code, taking advantage of the lack of side effects



## Contents


8 The $\lambda_0$ language

9 Abstract data types (ADTs)

10 Implementation

11 Recursion

12 Language specification

## The *Bool* ADT
Base contrsuctors and operators

- Base constructors:
  - $T : \to Bool$
  - $F : \to Bool$
- Operators:
  - $not : Bool \to Bool$
  - $and : Bool^2 \to Bool$
  - $or : Bool^2 \to Bool$
  - $if : Bool \times T \times T \to T$



## The *Bool* ADT
Axioms

- *not*
  - $(not\ \ T) = F$
  - $(not\ \ F) = T$
- *and*
  - $(and\ \ T\ \ a) = a$
  - $(and\ \ F\ \ a) = F$
- *or*
  - $(or\ \ T\ \ a) = T$
  - $(or\ \ F\ \ a) = a$
- *if*
  - $(if\ \ T\ \ a\ \ b) = a$
  - $(if\ \ F\ \ a\ \ b) = b$



## The *Bool* ADT
Base constructor implementation

- Intuition: selecting one of the two values, *true* or *false*
- $T \equiv_{def} \lambda xy.x$
- $F \equiv_{def} \lambda xy.y$
- Selector-like behavior:
  - $(T\ \ a\ \ b) \to (\lambda xy.x\ \ a\ \ b) \to a$
  - $(F\ \ a\ \ b) \to (\lambda xy.y\ \ a\ \ b) \to b$



## The *Bool* ADT
Operator implementation

- $not \equiv_{def} \lambda x.(x\ \ F\ \ T)$
  - $(not\ \ T) \to (\lambda x.(x\ \ F\ \ T)\ \ T) \to (T\ \ F\ \ T) \to F$
  - $(not\ \ F) \to (\lambda x.(x\ \ F\ \ T)\ \ F) \to (F\ \ F\ \ T) \to T$
- $and \equiv_{def} \lambda xy.(x\ \ y\ \ F)$
  - $(and\ \ T\ \ a) \to (\lambda xy.(x\ \ y\ \ F)\ \ T\ \ a) \to (T\ \ a\ \ F) \to a$
  - $(and\ \ F\ \ a) \to (\lambda xy.(x\ \ y\ \ F)\ \ F\ \ a) \to (F\ \ a\ \ F) \to F$
- $or \equiv_{def} \lambda xy.(x\ \ T\ \ y)$
  - $(or\ \ T\ \ a) \to (\lambda xy.(x\ \ T\ \ y)\ \ T\ \ a) \to (T\ \ T\ \ a) \to T$
  - $(or\ \ F\ \ a) \to (\lambda xy.(x\ \ T\ \ y)\ \ F\ \ a) \to (F\ \ T\ \ a) \to a$
- $if \equiv_{def} \lambda cte.(c\ \ t\ \ e)$ non-strict!
  - $(if\ \ T\ \ a\ \ b) \to (\lambda cte.(c\ \ t\ \ e)\ \ T\ \ a\ \ b) \to (T\ \ a\ \ b) \to a$
  - $(if\ \ F\ \ a\ \ b) \to (\lambda cte.(c\ \ t\ \ e)\ \ F\ \ a\ \ b) \to (F\ \ a\ \ b) \to b$



## The *Pair* ADT
Specification

- Base constructors:
  - $pair : A \times B \to Pair$
- Operators:
  - $fst : Pair \to A$
  - $snd : Pair \to B$
- Axioms:
  - $(fst\ \ (pair\ \ a\ \ b)) = a$
  - $(snd\ \ (pair\ \ a\ \ b)) = b$



## The *Pair* ADT
Implementation

- Intuition: a pair = a function that expects a selector, in order to apply it onto its components
- $pair \equiv_{def} \lambda xys.(s\ \ x\ \ y)$
  - $(pair\ \ a\ \ b) \to (\lambda xys.(s\ \ x\ \ y)\ \ a\ \ b) \to \lambda s.(s\ \ a\ \ b)$
- $fst \equiv_{def} \lambda p.(p\ \ T)$
  - $(fst\ \ (pair\ \ a\ \ b)) \to (\lambda p.(p\ \ T)\ \ \lambda s.(s\ \ a\ \ b)) \to$
    $(\lambda s.(s\ \ a\ \ b)\ \ T) \to (T\ \ a\ \ b) \to a$
- $snd \equiv_{def} \lambda p.(p\ \ F)$
  - $(snd\ \ (pair\ \ a\ \ b)) \to (\lambda p.(p\ \ F)\ \ \lambda s.(s\ \ a\ \ b)) \to$
    $(\lambda s.(s\ \ a\ \ b)\ \ F) \to (F\ \ a\ \ b) \to b$



## The *List* ADT
Base constructors and operators

- Base constructors:
  - $null : \to List$
  - $cons : A \times List \to List$
- Operators:
  - $car : List \setminus \{null\} \to A$
  - $cdr : List \setminus \{null\} \to List$
  - $null? : List \to Bool$
  - $append : List^2 \to List$



## The *List* ADT
Axioms

- *car*
  - $(car\ \ (cons\ \ e\ \ L)) = e$
- *cdr*
  - $(cdr\ \ (cons\ \ e\ \ L)) = L$
- *null?*
  - $(null?\ \ null) = T$
  - $(null?\ \ (cons\ \ e\ \ L)) = F$
- *append*
  - $(append\ \ null\ \ B) = B$
  - $(append\ \ (cons\ \ e\ \ A)\ \ B) = (cons\ \ e\ \ (append\ \ A\ \ B))$

## The *List* ADT
### Implementation

- Intuition: a list = a (head, tail) pair
- $null \equiv_{def} \lambda x.T$
- $cons \equiv_{def} pair$
- $car \equiv_{def} fst$
- $cdr \equiv_{def} snd$
- $null? \equiv_{def} \lambda L.(L\ \lambda xy.F)$
  - $(null?\ null) \to (\lambda L.(L\ \lambda xy.F)\ \lambda x.T) \to (\lambda x.T\ \ldots) \to T$
  - $(null?\ (cons\ e\ L)) \to (\lambda L.(L\ \lambda xy.F)\ \lambda s.(s\ e\ L)) \to (\lambda s.(s\ e\ L)\ \lambda xy.F) \to (\lambda xy.F\ e\ L) \to F$
- $append \equiv_{def}$ ... no closed form
  $\lambda AB.(if\ (null?\ A)\ B\ (cons\ (car\ A)\ (append\ (cdr\ A)\ B)))$



## The *Natural* ADT
### Axioms

- $zero?$
  - $(zero?\ zero) = T$
  - $(zero?\ (succ\ n)) = F$
- $pred$
  - $(pred\ (succ\ n)) = n$
- $add$
  - $(add\ zero\ n) = n$
  - $(add\ (succ\ m)\ n) = (succ\ (add\ m\ n))$



## The *Natural* ADT
### Implementation

- Intuition: a number = a list having the number value as its length
- $zero \equiv_{def} null$
- $succ \equiv_{def} \lambda n.(cons\ null\ n)$
- $zero? \equiv_{def} null?$
- $pred \equiv_{def} cdr$
- $add \equiv_{def} append$



## Contents


8 The $\lambda_0$ language
9 Abstract data types (ADTs)
10 Implementation
11 Recursion
12 Language specification




## Functions

- Several possible definitions of the identity function:
  - $id(n) = n$
  - $id(n) = n+1-1$
  - $id(n) = n+2-2$
  - ...
- Infinitely many textual representations for the same function
- Then... what is a function? A relation between inputs and outputs, independent of any textual representation e.g.,
  $id = \{(0,0),(1,1),(2,2),\ldots\}$



## Perspectives on recursion

- Textual: a function that refers itself, using its name
- Constructivist: recursive functions as values of an ADT, with specific ways of building them
- Semantic: the mathematical object designated by a recursive function



## Implementing *length*
### Problem

- Length of a list:

  $length \equiv_{def} \lambda L.(if\ (null?\ L)\ zero\ (succ\ (\underline{length}\ (cdr\ L))))$
- What do we replace the underlined area with, to avoid textual recursion?
- Rewrite the definition as a fixed-point equation

  $Length \equiv_{def} \lambda f L.(if\ (null?\ L)\ zero\ (succ\ (f\ (cdr\ L))))$
  $(Length\ length) \to length$
- How do we compute the fixed point? (see code archive)



## Contents


8 The $\lambda_0$ language
9 Abstract data types (ADTs)
10 Implementation
11 Recursion
12 Language specification

## Axiomatization benefits

- Disambiguation
- Proof of properties
- Implementation skeleton



## Syntax

- Variable:

$$Var ::= \text{any symbol distinct from } \lambda, ., (, )$$

- Expression:

$$\begin{aligned} Expr ::= &\ Var \\ \mid &\ \lambda Var.Expr \\ \mid &\ (Expr\ Expr) \end{aligned}$$

- Value:

$$Val ::= \lambda Var.Expr$$



## Evaluation rules

*Rule name*:

$$\frac{precondition_1, \ldots, precondition_n}{conclusion}$$



## Semantics for normal-order evaluation

### Evaluation

- *Reduce*:

$$(\lambda x.e\ e') \to e_{[e'/x]}$$

- *Eval*:

$$\frac{e \to e'}{(e\ e'') \to (e'\ e'')}$$



## Semantics for normal-order evaluation

### Substitution

- $x_{[e/x]} = e$
- $y_{[e/x]} = y, \quad y \neq x$
- $\langle \lambda x.e \rangle_{[e'/x]} = \lambda x.e$
- $\langle \lambda y.e \rangle_{[e'/x]} = \lambda y.e_{[e'/x]}, \quad y \neq x \wedge y \notin FV(e')$
- $\langle \lambda y.e \rangle_{[e'/x]} = \lambda z.e_{[z/y][e'/x]},$
  $y \neq x \wedge y \in FV(e') \wedge z \notin FV(e) \cup FV(e')$
- $(e'\ e'')_{[e/x]} = (e'_{[e/x]}\ e''_{[e/x]})$



## Semantics for normal-order evaluation

### Free variables

- $FV(x) = \{x\}$
- $FV(\lambda x.e) = FV(e) \setminus \{x\}$
- $FV((e'\ e'')) = FV(e') \cup FV(e'')$



## Semantics for normal-order evaluation

### Example

**Example 12.1 (Evaluation rules).**

$$((\lambda x.\lambda y.y\ a)\ b)$$

$$\frac{(\lambda x.\lambda y.y\ a) \to \lambda y.y \quad (Reduce)}{((\lambda x.\lambda y.y\ a)\ b) \to (\lambda y.y\ b)} \quad (Eval)$$

$$(\lambda y.y\ b) \to b \quad (Reduce)$$



## Semantics for applicative-order evaluation

### Evaluation

- *Reduce* ($v \in Val$):

$$(\lambda x.e\ v) \to e_{[v/x]}$$

- $Eval_1$:

$$\frac{e \to e'}{(e\ e'') \to (e'\ e'')}$$

- $Eval_2$ ($v \in Val$):

$$\frac{e \to e'}{(v\ e) \to (v\ e')}$$

## Formal proof

**Proposition 12.2 (Free and bound variables).**

$$\forall e \in Expr \bullet BV(e) \cap FV(e) = \emptyset$$

Proof.
Structural induction, according to the different forms of $\lambda$-expressions (see the lecture notes). □



## Summary

- Practical usage of the untyped lambda calculus, as a programming language
- Formal specifications, for different evaluation semantics



# Part IV

# Typed Lambda Calculus



## Contents


13 Introduction

14 Simply Typed Lambda Calculus (STLC, System $F_1$)

15 Extending STLC

16 Polymorphic Lambda Calculus (PSTLC, System F)

17 Type reconstruction

18 Higher-Order Polymorphic Lambda Calculus (HPSTLC, System $F_\omega$)




## Contents


13 Introduction

14 Simply Typed Lambda Calculus (STLC, System $F_1$)

15 Extending STLC

16 Polymorphic Lambda Calculus (PSTLC, System F)

17 Type reconstruction

18 Higher-Order Polymorphic Lambda Calculus (HPSTLC, System $F_\omega$)




## Drawbacks of the absence of types

- Meaningless expressions e.g., $(car\ 3)$
- No canonical representation for the values of a given type e.g., both a tree and a set having the same representation
- Impossibility of translating certain expressions into certain typed languages e.g., $(x\ x)$, $\Omega$, $Fix$
- Potential irreducibility of expressions — inconsistent representation of equivalent values

$$\lambda x.(Fix\ x) \to \lambda x.(x\ (Fix\ x)) \to \lambda x.(x\ (x\ (Fix\ x))) \to \ldots$$



## Solution

- Restricted ways of constructing expressions, depending on the types of their parts
- Sacrificed expressivity in change for soundness



## Desired properties

**Definition 13.1 (Progress).**
A well-typed expression is either a value or is subject to at least one reduction step.

**Definition 13.2 (Preservation).**
The result obtained by reducing a well-typed expression is well-typed. Usually, the type is the same.

**Definition 13.3 (Strong normalization).**
The evaluation of a well-typed expression terminates.

## Contents


13 Introduction

14 Simply Typed Lambda Calculus (STLC, System $F_1$)

15 Extending STLC

16 Polymorphic Lambda Calculus (PSTLC, System F)

17 Type reconstruction

18 Higher-Order Polymorphic Lambda Calculus (HPSTLC, System $F_\omega$)




## Base and simple types

**Definition 14.1 (Base type).**
An atomic type e.g., numbers, booleans etc.

**Definition 14.2 (Simple type).**
A type built from existing types e.g., $\sigma \to \tau$, where $\sigma$ and $\tau$ are types.

Notation:

- $e : \tau$: "expression $e$ has type $\tau$"
- $v \in \tau$: "$v$ is a value of type $\tau$"
- $e \in \tau \Rightarrow e : \tau$
- $e : \tau \not\Rightarrow e \in \tau$



## Typed $\lambda$-expressions

**Definition 14.3 ($\lambda_t$-expression).**

- Base value: a base value $b \in \tau_b$ is a $\lambda_t$-expression.
- Typed variable: an (explicitly) typed variable $x : \tau$ is a $\lambda_t$-expression.
- Function: if $x : \sigma$ is a typed variable and $e : \tau$ is a $\lambda_t$-expression, then $\lambda x : \sigma.e : \sigma \to \tau$ is a $\lambda_t$-expression, which stands for ....
- Application: if $f : \sigma \to \tau$ and $a : \sigma$ are $\lambda_t$-expressions, then $(f\ \ a) : \tau$ is a $\lambda_t$-expression, which stands for ....



## Relation to untyped lambda calculus

Similarities

- $\beta$-reduction
- $\alpha$-conversion
- normal forms
- Church-Rosser theorem

Differences

- $(x : \tau\ \ x : \tau)$ invalid
- some fixed-point combinators are invalid



## Syntax
### Expressions

- Variables:

$$Var ::= \ldots$$

- Expressions:

$$\begin{aligned} Expr &::= Val \\ &\mid Var \\ &\mid (Expr\ \ Expr) \end{aligned}$$

- Values:

$$\begin{aligned} Val &::= BaseVal \\ &\mid \lambda Var : Type.Expr \end{aligned}$$



## Syntax
### Types

- Types:

$$\begin{aligned} Type &::= BaseType \\ &\mid (Type \to Type) \end{aligned}$$

- Typing contexts:
  - include variable-type associations i.e., *typing hypotheses*

$$\begin{aligned} TypingContext &::= \emptyset \\ &\mid TypingContext, Var : Type \end{aligned}$$



## Semantics for normal-order evaluation
### Evaluation

- *Reduce*:

$$(\lambda x : \tau.e\ \ e') \to e_{[e'/x]}$$

- *Eval*:

$$\frac{e \to e'}{(e\ \ e'') \to (e'\ \ e'')}$$

The type annotations are ignored, since typing precedes evaluation.



## Semantics
### Typing

- *TBaseVal*:

$$\frac{v \in \tau_b}{\Gamma \vdash v : \tau_b}$$

- *TVar*:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

- *TAbs*:

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.e : (\tau \to \tau')}$$

- *TApp*:

$$\frac{\Gamma \vdash e : (\tau' \to \tau) \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e\ \ e') : \tau}$$

## Typing example

**Example 14.4 (Typing).**

$$\lambda x:\tau_1.\lambda y:\tau_2.x:(\tau_1\to(\tau_2\to\tau_1))$$

Blackboard!



## Type systems

**Definition 14.5 (Type system).**
The set of rules and mechanisms used in a programming language to organize, build and handle the types accepted in the language.

**Definition 14.6 (Soundness).**
The type system of a language is *sound* if any well-typed expression in the language has the progress and preservation properties.

**Proposition 14.7.**
*STLC is sound and possesses the strong normalization property.*



## Contents

13 Introduction
14 Simply Typed Lambda Calculus (STLC, System $F_1$)
15 Extending STLC
16 Polymorphic Lambda Calculus (PSTLC, System F)
17 Type reconstruction
18 Higher-Order Polymorphic Lambda Calculus (HPSTLC, System $F_\omega$)



## Ways of extending STLC

1. Particular base types
2. $n$-ary type constructors, $n \geq 1$, which build simple types



## The product type

### Algebraic specification

- Base constructors i.e., canonical values:
  - $\tau * \tau' ::= (\tau, \tau')$
- Operators:
  - $fst : \tau * \tau' \to \tau$
  - $snd : \tau * \tau' \to \tau'$
- Axioms ($e : \tau$, $e' : \tau'$):
  - $(fst\ (e, e')) \to e$
  - $(snd\ (e, e')) \to e'$



## The product type

### Syntax

$$\begin{array}{lcl} Expr & ::= & \ldots \\ & \mid & (fst\ Expr) \\ & \mid & (snd\ Expr) \\ & \mid & (Expr, Expr) \\ BaseVal & ::= & \ldots \\ & \mid & ProductVal \\ ProductVal & ::= & (Val, Val) \\ Type & ::= & \ldots \\ & \mid & (Type * Type) \end{array}$$



## The product type

### Evaluation

- *EvalFst*:
$$(fst\ (e, e')) \to e$$
- *EvalSnd*:
$$(snd\ (e, e')) \to e'$$
- *EvalFstApp*:
$$\frac{e \to e'}{(fst\ e) \to (fst\ e')}$$
- *EvalSndApp*:
$$\frac{e \to e'}{(snd\ e) \to (snd\ e')}$$



## The product type

### Typing

- *TProduct*:
$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e, e') : (\tau * \tau')}$$
- *TFst*:
$$\frac{\Gamma \vdash e : (\tau * \tau')}{\Gamma \vdash (fst\ e) : \tau}$$
- *TSnd*:
$$\frac{\Gamma \vdash e : (\tau * \tau')}{\Gamma \vdash (snd\ e) : \tau'}$$

## The product type
### Typing example

**Example 15.1 (Typing).**

$$\Gamma \vdash \lambda x : ((\rho * \tau) \to \sigma).\lambda y : \rho.\lambda z : \tau.(x\ \ (y, z))$$
$$: ((\rho * \tau) \to \sigma) \to \rho \to \tau \to \sigma$$

Blackboard!



## The *Bool* type
### Algebraic specification

- Base constructors i.e., canonical values:
  - $Bool ::= True \mid False$
- Operators:
  - $not : Bool \to Bool$
  - $and : Bool^2 \to Bool$
  - $or : Bool^2 \to Bool$
  - $if : Bool \times \tau \times \tau \to \tau$
- Axioms: see slide 81



## The *Bool* type
### Syntax

$$\begin{aligned} Expr &::= \ \ldots \\ &\mid \ (if\ \ Expr\ \ Expr\ \ Expr) \end{aligned}$$

$$\begin{aligned} BaseVal &::= \ \ldots \\ &\mid \ BoolVal \end{aligned}$$

$$BoolVal ::= True \mid False$$

$$\begin{aligned} BaseType &::= \ \ldots \\ &\mid \ Bool \end{aligned}$$



## The *Bool* type
### Evaluation

- *EvalIfT*:
$$(if\ \ True\ \ e\ \ e') \to e$$
- *EvalIfF*:
$$(if\ \ False\ \ e\ \ e') \to e'$$
- *EvalIf*:
$$\frac{e \to e'}{(if\ \ e\ \ e_1\ \ e_2) \to (if\ \ e'\ \ e_1\ \ e_2)}$$



## The *Bool* type
### Typing

- *TTrue*:
$$\Gamma \vdash True : Bool$$
- *TFalse*:
$$\Gamma \vdash False : Bool$$
- *TIf*:
$$\frac{\Gamma \vdash e : Bool \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (if\ \ e\ \ e_1\ \ e_2) : \tau}$$



## The *Bool* type
### Top-level variable bindings

- $not \equiv \lambda x : Bool.(if\ \ x\ \ False\ \ True)$
- $and \equiv \lambda x : Bool.\lambda y : Bool.(if\ \ x\ \ y\ \ False)$
- $or \equiv \lambda x : Bool.\lambda y : Bool.(if\ \ x\ \ True\ \ y)$



## The $\mathbb{N}$ type
### Algebraic specification

- Base constructors i.e., canonical values:
  - $\mathbb{N} ::= 0 \mid (succ\ \ \mathbb{N})$
- Operators:
  - $+ : \mathbb{N}^2 \to \mathbb{N}$
  - $zero? : \mathbb{N} \to Bool$
- Axioms ($m, n \in \mathbb{N}$):
  - $(+\ \ 0\ \ n) = n$
  - $(+\ \ (succ\ \ m)\ \ n) = (succ\ \ (+\ \ m\ \ n))$
  - $(zero?\ \ 0) = True$
  - $(zero?\ \ (succ\ \ n)) = False$



## The $\mathbb{N}$ type
### Operator semantics

- How to avoid defining evaluation and typing rules for each operator of $\mathbb{N}$?
- Introduce the primitive recursor for $\mathbb{N}$, $prec_{\mathbb{N}}$, which allows for defining any primitive recursive function on natural numbers
- Define the operators using the primitive recursor

## The $\mathbb{N}$ type
### Syntax

$$\begin{array}{lcl} Expr & ::= & \ldots \\ & | & (succ\ Expr) \\ & | & (prec_{\mathbb{N}}\ Expr\ Expr\ Expr) \end{array}$$

$$\begin{array}{lcl} BaseVal & ::= & \ldots \\ & | & NVal \end{array}$$

$$\begin{array}{lcl} NVal & ::= & 0 \\ & | & (succ\ NVal) \end{array}$$

$$\begin{array}{lcl} BaseType & ::= & \ldots \\ & | & \mathbb{N} \end{array}$$



## The $\mathbb{N}$ type
### Evaluation

- *EvalSucc*:
$$\frac{e \rightarrow e'}{(succ\ e) \rightarrow (succ\ e')}$$
- $EvalPrec_{\mathbb{N}0}$:
$$(prec_{\mathbb{N}}\ e_0\ f\ 0) \rightarrow e_0$$
- $EvalPrec_{\mathbb{N}1}$ $(n \in \mathbb{N})$:
$$(prec_{\mathbb{N}}\ e_0\ f\ (succ\ n)) \rightarrow (f\ n\ (prec_{\mathbb{N}}\ e_0\ f\ n))$$
- $EvalPrec_{\mathbb{N}2}$:
$$\frac{e \rightarrow e'}{(prec_{\mathbb{N}}\ e_0\ f\ e) \rightarrow (prec_{\mathbb{N}}\ e_0\ f\ e')}$$



## The $\mathbb{N}$ type
### Typing

- *TZero*:
$$\Gamma \vdash 0 : \mathbb{N}$$
- *TSucc*:
$$\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash (succ\ e) : \mathbb{N}}$$
- $TPrec_{\mathbb{N}}$:
$$\frac{\Gamma \vdash e_0 : \tau \qquad \Gamma \vdash f : \mathbb{N} \rightarrow \tau \rightarrow \tau \qquad \Gamma \vdash e : \mathbb{N}}{\Gamma \vdash (prec_{\mathbb{N}}\ e_0\ f\ e) : \tau}$$



## The $\mathbb{N}$ type
### Top-level variable bindings

- $zero? \equiv \lambda n : \mathbb{N}.(prec_{\mathbb{N}}\ True\ \lambda x : \mathbb{N}.\lambda y : Bool.False\ n)$
- $+ \equiv \lambda m : \mathbb{N}.\lambda n : \mathbb{N}.(prec_{\mathbb{N}}\ n\ \lambda x : \mathbb{N}.\lambda y : \mathbb{N}.(succ\ y)\ m)$



## The $(List\ \tau)$ type
### Algebraic specification

- Base constructors i.e., canonical values:
  - $(List\ \tau) \quad ::= \quad []_{\tau} \quad | \quad (cons\ \tau\ (List\ \tau))$
- Operators:
  - $head : (List\ \tau) \setminus \{[]\} \rightarrow \tau$
  - $tail : (List\ \tau) \setminus \{[]\} \rightarrow (List\ \tau)$
  - $length : (List\ \tau) \rightarrow \mathbb{N}$
- Axioms $(h \in \tau,\ t \in (List\ \tau))$:
  - $(head\ (cons\ h\ t)) = h$
  - $(tail\ (cons\ h\ t)) = t$
  - $(length\ []) = 0$
  - $(length\ (cons\ h\ t)) = (succ\ (length\ t))$



## The $(List\ \tau)$ type
### Syntax

$$\begin{array}{lcl} Expr & ::= & \ldots \\ & | & (cons\ Expr\ Expr) \\ & | & (prec_L\ Expr\ Expr\ Expr) \end{array}$$

$$\begin{array}{lcl} BaseVal & ::= & \ldots \\ & | & ListVal \end{array}$$

$$\begin{array}{lcl} ListVal & ::= & [] \\ & | & (cons\ Value\ ListVal) \end{array}$$

$$\begin{array}{lcl} Type & ::= & \ldots \\ & | & (List\ Type) \end{array}$$



## The $(List\ \tau)$ type
### Evaluation

- *EvalCons*:
$$\frac{e \rightarrow e'}{(cons\ e\ e'') \rightarrow (cons\ e'\ e'')}$$
- $EvalPrec_{L0}$:
$$(prec_L\ e_0\ f\ []) \rightarrow e_0$$
- $EvalPrec_{L1}$ $(v \in Value)$:
$$(prec_L\ e_0\ f\ (cons\ v\ e)) \rightarrow (f\ v\ e\ (prec_L\ e_0\ f\ e))$$
- $EvalPrec_{L2}$:
$$\frac{e \rightarrow e'}{(prec_L\ e_0\ f\ e) \rightarrow (prec_L\ e_0\ f\ e')}$$



## The $(List\ \tau)$ type
### Typing

- *TEmpty*:
$$\Gamma \vdash []_{\tau} : (List\ \tau)$$
- *TCons*:
$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e' : (List\ \tau)}{\Gamma \vdash (cons\ e\ e') : (List\ \tau)}$$
- $TPrec_L$:
$$\frac{\Gamma \vdash e_0 : \tau' \qquad \Gamma \vdash f : \tau \rightarrow (List\ \tau) \rightarrow \tau' \rightarrow \tau' \qquad \Gamma \vdash e : (List\ \tau)}{\Gamma \vdash (prec_L\ e_0\ f\ e) : \tau'}$$

## The $(List\ \tau)$ type
Top-level variable bindings

- $empty? \equiv \lambda l : (List\ \tau).(prec_L\ True\ f\ l)$,
  $f \equiv \lambda h : \tau.\lambda t : (List\ \tau).\lambda r : Bool.False$
- $length \equiv \lambda l : (List\ \tau).(prec_L\ 0\ f\ l)$,
  $f \equiv \lambda h : \tau.\lambda t : (List\ \tau).\lambda r : \mathbb{N}.(succ\ r)$



## General recursion

- Primitive recursion
  - induces *strong normalization*
  - insufficient for capturing effectively computable functions
- Introduce the operator *fix* i.e., a fixed-point combinator
- Gain computation power at the expense of strong normalization



## *fix*
Syntax

$$Expr ::= \ldots \mid (fix\ Expr)$$



## *fix*
Evaluation

- *EvalFix*:
  $$(fix\ \lambda x : \tau.e) \rightarrow e_{[(fix\ \lambda x:\tau.e)/x]} = (f\ (fix\ f))$$
- *EvalFix'*:
  $$\frac{e \rightarrow e'}{(fix\ e) \rightarrow (fix\ e')}$$



## *fix*
Typing

- *TFix*:
  $$\frac{\Gamma \vdash e : (\tau \rightarrow \tau)}{\Gamma \vdash (fix\ e) : \tau}$$



## *fix*
Example

**Example 15.2 (The *remainder* function).**

$$remainder = \lambda m : \mathbb{N}.\lambda n : \mathbb{N}.\ ((fix\ \lambda f : (\mathbb{N} \rightarrow \mathbb{N}).\lambda p : \mathbb{N}.\ (if\ p < n\ then\ p\ else\ (f\ (p - n))))\ m)$$

The evaluation of $(remainder\ 3\ 0)$ does not terminate.



## Monomorphism

- Within the types $(\tau * \tau')$ and $(List\ \tau)$, $\tau$ and $\tau'$ designate specific types e.g., *Bool*, $\mathbb{N}$, $(List\ \mathbb{N})$, etc.
- Dedicated operators for each simple type
- $fst_{\mathbb{N},Bool}$, $fst_{Bool,\mathbb{N}}$, ...
- $[]_{\mathbb{N}}$, $[]_{Bool}$, ...
- $empty?_{\mathbb{N}}$, $empty?_{Bool}$, ...



## Contents

13 Introduction
14 Simply Typed Lambda Calculus (STLC, System $F_1$)
15 Extending STLC
16 Polymorphic Lambda Calculus (PSTLC, System F)
17 Type reconstruction
18 Higher-Order Polymorphic Lambda Calculus (HPSTLC, System $F_\omega$)

## Idea

- Monomorphic identity function for type $\mathbb{N}$:

$$id_{\mathbb{N}} \equiv \lambda x : \mathbb{N}.x : (\mathbb{N} \to \mathbb{N})$$

- Polymorphic identity function — type variables:

$$id \equiv \lambda X.\lambda x : \mathbb{N}.x : \forall X.(X \to X)$$

- Type coercion prior to function application:

$$(id[\mathbb{N}]\ 5) \to (id_{\mathbb{N}}\ 5) \to 5$$



## Syntax

- Program variables: stand for program values

$$Var \quad ::= \quad \ldots$$

- Type variables: stand for types

$$TypeVar \quad ::= \quad \ldots$$



## Syntax

- Expressions:

$$\begin{array}{lll} Expr & ::= & Value \\ & \mid & Var \\ & \mid & (Expr\ Expr) \\ & \mid & Expr[Type] \end{array}$$

- Values:

$$\begin{array}{lll} Value & ::= & BaseValue \\ & \mid & \lambda Var : Type.Expr \\ & \mid & \lambda TypeVar.Expr \end{array}$$



## Syntax

- Types:

$$\begin{array}{lll} Type & ::= & BaseType \\ & \mid & TypeVar \\ & \mid & (Type \to Type) \\ & \mid & \forall TypeVar.Type \end{array}$$

- Typing contexts:

$$\begin{array}{lll} TypingContext & ::= & \emptyset \\ & \mid & TypingContext, Var : Type \\ & \mid & TypingContext, TypeVar \end{array}$$



## Semantics
### Evaluation

- $Reduce_1$:

$$(\lambda x : \tau.e\ e') \to e_{[e'/x]}$$

- $Reduce_2$:

$$\lambda X.e[\tau] \to e_{[\tau/X]}$$

- $Eval_1$:

$$\frac{e \to e'}{(e\ e'') \to (e'\ e'')}$$

- $Eval_2$:

$$\frac{e \to e'}{e[\tau] \to e'[\tau]}$$



## Semantics
### Typing

- $TBaseValue$:

$$\frac{v \in \tau_b}{\Gamma \vdash v : \tau_b}$$

- $TVar$:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

- $TAbs_1$:

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.e : (\tau \to \tau')}$$

- $TApp_1$:

$$\frac{\Gamma \vdash e : (\tau' \to \tau) \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e\ e') : \tau}$$



## Semantics
### Typing

- $TAbs_2$ — polymorphic expressions have universal types:

$$\frac{\Gamma, X \vdash e : \tau}{\Gamma \vdash \lambda X.e : \forall X.\tau}$$

- $TApp_2$:

$$\frac{\Gamma \vdash e : \forall X.\tau}{\Gamma \vdash e[\tau'] : \tau_{[\tau'/X]}}$$



## Semantics
### Substitution and free variables

- $Expr_{[Expr/Var]}$
- $Expr_{[Type/TypeVar]}$
- $Type_{[Type/TypeVar]}$
- Free program variables
- Free type variables

## Typing example

**Example 16.1 (Typing).**

$$\Gamma \vdash \lambda f : \forall X.(X \to X).\lambda Y.\lambda x : Y.(f[Y]\ x)$$
$$: (\forall X.(X \to X) \to \forall Y.(Y \to Y))$$

Monomorphic function
with polymorphic argument and result!

Blackboard!



## Examples of polymorphic expressions

**Example 16.2 (Doubling a computation).**

$$\begin{array}{rcl} double & \equiv & \lambda X.\lambda f : (X \to X).\lambda x : X.(f\ (f\ x)) \\ & : & \forall X.((X \to X) \to (X \to X)) \end{array}$$

**Example 16.3 (Quadrupling a computation).**

$$\begin{array}{rcl} quadruple & \equiv & \lambda X.(double[X \to X]\ double[X]) \\ & : & \forall X.((X \to X) \to (X \to X)) \end{array}$$



## Examples of polymorphic expressions

**Example 16.4 (Reflexive computation).**

$$\begin{array}{rcl} reflexive & \equiv & \lambda f : \forall X.(X \to X).(f[\forall X.(X \to X)]\ f) \\ & : & (\forall X.(X \to X) \to \forall X.(X \to X)) \end{array}$$

**Example 16.5 (Fixed-point combinator).**

$$\begin{array}{rcl} Fix & \equiv & \lambda X.\lambda f : (X \to X).(f\ (Fix[X]\ f)) \\ & : & \forall X.((X \to X) \to X) \end{array}$$



## Contents


13 Introduction

14 Simply Typed Lambda Calculus (STLC, System $F_1$)

15 Extending STLC

16 Polymorphic Lambda Calculus (PSTLC, System F)

17 Type reconstruction

18 Higher-Order Polymorphic Lambda Calculus (HPSTLC, System $F_\omega$)




## Motivation



## Contents


13 Introduction

14 Simply Typed Lambda Calculus (STLC, System $F_1$)

15 Extending STLC

16 Polymorphic Lambda Calculus (PSTLC, System F)

17 Type reconstruction

18 Higher-Order Polymorphic Lambda Calculus (HPSTLC, System $F_\omega$)




## Problem

- Polymorphic identity function, on objects of a type built using 1-ary type constructors e.g., *List*:

  $$f \equiv \lambda C.\lambda X.\lambda x : (C\ X).x : \forall C.\forall X.((C\ X) \to (C\ X))$$

- $C$ stands for a 1-ary type constructor, $X$ stands for a type of program values i.e., a *proper type*
- Monomorphic identity function for type $(List\ \mathbb{N})$:

  $$f[List][\mathbb{N}] \to \lambda x : (List\ \mathbb{N}).x : ((List\ \mathbb{N}) \to (List\ \mathbb{N}))$$

- How do we prevent erroneous situations e.g., $f[\mathbb{N}][\mathbb{N}]$, $f[List][List]$?



## Solution

- Two categories of types: proper types, and type constructors i.e., $\lambda\, TypeVar.Type$
- Type not only program variables, but also type variables
- The type of a type: *kind*

## Kinds
### Notation

- The kind of a proper type: $*$
- The kind of a 1-ary type constructor: $(* \Rightarrow *)$
- The kind of an $n$-ary type constructor, $n \geq 1$: $k_1 \Rightarrow k_2$
- The kind $k$ of a type $\tau$: $\tau :: k$



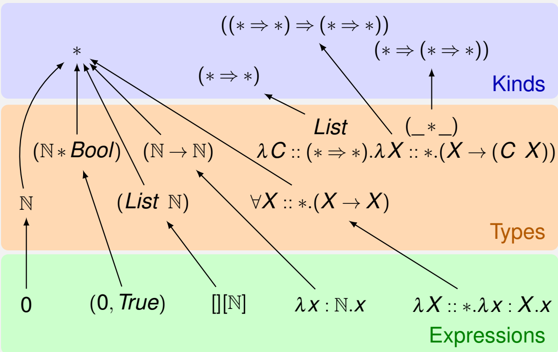
## Kinds
### Examples

**Example 18.1 (Kinds).**

- $\mathbb{N} :: *$
- $List :: (* \Rightarrow *)$
- $f \equiv \lambda C :: (* \Rightarrow *).\lambda X :: *.\lambda x : (C\ X).x$
  $f : \forall C :: (* \Rightarrow *).\forall X :: *.((C\ X) \rightarrow (C\ X))$



## Levels of expressions






## Type equivalence

- Two syntactically distinct types:

  $\tau_1 \equiv ((List\ \mathbb{N}) \rightarrow (List\ \mathbb{N}))$
  $\tau_2 \equiv (\lambda X :: *.((List\ X) \rightarrow (List\ X))\ \mathbb{N})$

- Semantically, they denote the same type i.e., they are equivalent: $\tau_1 \equiv \tau_2$



## Syntax

- Expressions:

$$\begin{array}{lcl} Expr & ::= & Value \\ & | & Var \\ & | & (Expr\ \ Expr) \\ & | & Expr[Type] \end{array}$$

- Values:

$$\begin{array}{lcl} Value & ::= & BaseValue \\ & | & \lambda\, Var : Type.Expr \\ & | & \lambda\, TypeVar :: Kind.Expr \end{array}$$



## Syntax

- Types:

$$\begin{array}{lcl} Type & ::= & BaseType \\ & | & TypeVar \\ & | & (Type \rightarrow Type) \\ & | & \forall\, TypeVar :: Kind.Type \\ & | & \lambda\, TypeVar :: Kind.Type \\ & | & (Type\ \ Type) \end{array}$$

- Typing contexts:

$$\begin{array}{lcl} TypingContext & ::= & \emptyset \\ & | & TypingContext, Var : Type \\ & | & TypingContext, TypeVar :: Kind \end{array}$$



## Syntax

- Kinds:

$$\begin{array}{lcl} Kind & ::= & * \\ & | & (Kind \Rightarrow Kind) \end{array}$$



## Semantics
### Evaluation

- $Reduce_1$:
$$(\lambda x : \tau.e\ \ e') \rightarrow e_{[e'/x]}$$
- $Reduce_2$:
$$\lambda X :: K.e[\tau] \rightarrow e_{[\tau/X]}$$
- $Eval_1$:
$$\frac{e \rightarrow e'}{(e\ \ e'') \rightarrow (e'\ \ e'')}$$
- $Eval_2$:
$$\frac{e \rightarrow e'}{e[\tau] \rightarrow e'[\tau]}$$

## Semantics
### Typing

- *TBaseValue*:
$$\frac{v \in \tau_b}{\Gamma \vdash v : \tau_b}$$
- *TVar*:
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$
- *TAbs*$_1$:
$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : (\tau \to \tau')}$$
- *TApp*$_1$:
$$\frac{\Gamma \vdash e : (\tau' \to \tau) \qquad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e\ e') : \tau}$$



## Semantics
### Typing

- *TAbs*$_2$:
$$\frac{\Gamma, X :: K \vdash e : \tau}{\Gamma \vdash \lambda X :: K.e : \forall X :: K.\tau}$$
- *TApp*$_2$:
$$\frac{\Gamma \vdash e : \forall X :: K.\tau \qquad \Gamma \vdash \tau' :: K}{\Gamma \vdash e[\tau'] : \tau_{[\tau'/X]}}$$



## Semantics
### Kinding

- *KBaseType*:
$$\Gamma \vdash \tau_b :: *$$
- *KTypeVar*:
$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K}$$
- *KTypeAbs*:
$$\frac{\Gamma, X :: K \vdash \tau :: K'}{\Gamma \vdash \lambda X :: K.\tau :: (K \Rightarrow K')}$$
- *KTypeApp*:
$$\frac{\Gamma \vdash \tau :: (K' \Rightarrow K) \qquad \Gamma \vdash \tau' :: K'}{\Gamma \vdash (\tau\ \tau') :: K}$$



## Semantics
### Kinding

- *KAbs*$_1$:
$$\frac{\Gamma \vdash \tau :: * \qquad \Gamma \vdash \tau' :: *}{\Gamma \vdash (\tau \to \tau') :: *}$$
- *KAbs*$_2$:
$$\frac{\Gamma, X :: K \vdash \tau :: *}{\Gamma \vdash \forall X :: K.\tau :: *}$$



## Semantics
### Type equivalence

- *EqReflexivity*:
$$\tau \equiv \tau$$
- *EqSymmetry*:
$$\frac{\tau \equiv \tau'}{\tau' \equiv \tau}$$
- *EqTransitivity*:
$$\frac{\tau \equiv \tau' \qquad \tau' \equiv \tau''}{\tau \equiv \tau''}$$
- *EqTypeReduce*:
$$(\lambda X :: K.\tau\ \tau') \equiv \tau_{[\tau'/X]}$$



## Semantics
### Type equivalence

- *EqTypeAbs*:
$$\frac{\tau \equiv \tau'}{\lambda X :: K.\tau \equiv \lambda X :: K.\tau'}$$
- *EqTypeApp*:
$$\frac{\tau \equiv \tau' \qquad \sigma \equiv \sigma'}{(\tau\ \sigma) \equiv (\tau'\ \sigma')}$$
- *EqAbs*$_1$:
$$\frac{\tau \equiv \tau' \qquad \sigma \equiv \sigma'}{(\tau \to \sigma) \equiv (\tau' \to \sigma')}$$
- *EqAbs*$_2$:
$$\frac{\tau \equiv \tau'}{\forall X :: K.\tau \equiv \forall X :: K.\tau'}$$



## Semantics
### Type equivalence

- *TypeEquivalence*:
$$\frac{\Gamma \vdash e : \tau \qquad \tau \equiv \tau'}{\Gamma \vdash e : \tau'}$$



## Kinding example

**Example 18.2 (Kinding).**

$$\forall X :: *.(X \to ((List\ \ X) \to (Tree\ \ X))) :: *$$

Blackboard!

Part V

# Constructive Type Theory



## Contents

19 Constructive paradigm

20 Syntax and semantics



## Contents

19 Constructive paradigm

20 Syntax and semantics



## Classical logic

- Example: prove $\exists x.P(x)$
- Perhaps, proof by contradiction: assume $\neg\exists x.P(x)$ and reach a contradiction
- Assumption: $\exists x.P(x) \vee \neg\exists x.P(x)$ (law of excluded middle)
- Problem: possibly no actual evidence regarding either sentence i.e., some $a$ s.t. either $P(a)$ or $\neg P(a)$ is true



## Constructive logic

- Prove $\exists x.P(x)$ by computing an object $a$ s.t. $P(a)$ is true
- Not always possible
- However, not being able to compute $a$ does not mean that $\exists x.P(x)$ is false
- Law of excluded middle — not an axiom in constructive logic

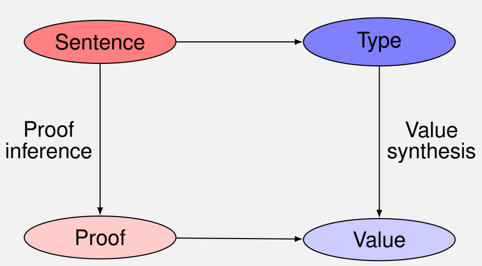


## Constructive type theory

- Bridge between constructive logic and typed lambda calculus
- Correspondences:
  - sentence $\leftrightarrow$ type
  - logical connective $\leftrightarrow$ type constructor
  - proof $\leftrightarrow$ function with that type
- Application: synthesize a program by proving the sentence that corresponds to its specification



## The Curry-Howard isomorphism






## Contents

19 Constructive paradigm

20 Syntax and semantics

## Two views

$$a : A$$

- Type-theoretic: "$a$ is a value of type $A$"
- Logical: "$a$ is a proof of sentence $A$"



## Definitional rules

| Rule | Logical view | Type-theoretic view |
|---|---|---|
| **Formation** | How a connective relates two sentences | How a type constructor is used |
| **Introduction/ elimination** | How a proof is derived | How a value is constructed |
| **Computation** | How a proof is simplified | How an expression is evaluated |



## Other logic-type correspondences

| Logical view | Type-theoretic view |
|---|---|
| Truth ($\top$) | One-element type, containing the trivial proof |
| Falsity ($\bot$) | No-element type, containing no proof |
| Proof by induction | Definition by recursion |



## Logical conjunction / product type constructor I

- Formation rule ($\wedge F$):

$$\frac{A \text{ is a sentence/ type} \qquad B \text{ is a sentence/ type}}{A \wedge B \text{ is a sentence/ type}}$$

- Introduction rule ($\wedge I$):

$$\frac{a : A \qquad b : B}{(a, b) : A \wedge B}$$



## Logical conjunction / product type constructor II

- Elimination rules ($\wedge E_{1,2}$):

$$\frac{p : A \wedge B}{fst\ \ p : A} \qquad \frac{p : A \wedge B}{snd\ \ p : B}$$

- Computation rules:

$$fst\ \ (a, b) \rightarrow a$$
$$snd\ \ (a, b) \rightarrow b$$



## Logical implication / function type constructor I

- Formation rule ($\Rightarrow F$):

$$\frac{A \text{ is a sentence/ type} \qquad B \text{ is a sentence/ type}}{A \Rightarrow B \text{ is a sentence/ type}}$$

- Introduction rule ($\Rightarrow I$)
  (square brackets = discharged assumption):

$$\frac{\begin{array}{c}[x : A] \\ \vdots \\ b : B\end{array}}{\lambda x : A.b : A \Rightarrow B}$$



## Logical implication / function type constructor II

- Elimination rule ($\Rightarrow E$):

$$\frac{a : A \qquad f : A \Rightarrow B}{(f\ \ a) : B}$$

- Computation rule:

$$(\lambda x : A.b\ \ a) \rightarrow b_{[a/x]}$$



## Logical disjunction / sum type constructor I

- Formation rule ($\vee F$):

$$\frac{A \text{ is a sentence/ type} \qquad B \text{ is a sentence/ type}}{A \vee B \text{ is a sentence/ type}}$$

- Introduction rules ($\vee I_{1,2}$):

$$\frac{a : A}{inl\ \ a : A \vee B} \qquad \frac{b : B}{inr\ \ b : A \vee B}$$

## Logical disjunction / sum type constructor II

- Elimination rule ($\vee E$):

$$\frac{p : A \vee B \qquad f : A \Rightarrow C \qquad g : B \Rightarrow C}{cases\ p\ f\ g : C}$$

- Computation rules:

$$cases\ (inl\ a)\ f\ g \to f\ a$$
$$cases\ (inr\ b)\ f\ g \to g\ b$$



## Absurd sentence / empty type I

- Formation rule ($\perp F$):

$$\frac{}{\perp \text{ is a sentence/ type}}$$

- Introduction rule: none — there is no proof of the absurd sentence



## Absurd sentence / empty type II

- Elimination rule ($\perp E$)
(a proof of the absurd sentence can prove anything):

$$\frac{p : \perp}{abort_A\ p : A}$$

- Computation rule: none



## Logical negation and equivalence

- Logical negation:

$$\neg A \equiv A \Rightarrow \perp$$

- Logical equivalence:

$$A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A)$$



## Example proofs

- $A \Rightarrow A$
- $A \Rightarrow \neg\neg A$ (converse?)
- $((A \wedge B) \Rightarrow C) \Rightarrow A \Rightarrow B \Rightarrow C$
- $(A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$
- $(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$
- $(A \vee B) \Rightarrow \neg(\neg A \wedge \neg B)$



## Universal quantification / generalized function type constructor I

- Formation rule ($\forall F$)
(square brackets = discharged assumption):

$$\frac{A \text{ is a sentence/ type} \qquad \begin{matrix}[x : A] \\ \vdots \\ B \text{ is a sentence/ type}\end{matrix}}{(\forall x : A).B \text{ is a sentence/ type}}$$

- Introduction rule ($\forall I$):

$$\frac{\begin{matrix}[x : A] \\ \vdots \\ b : B\end{matrix}}{(\lambda x : A).b : (\forall x : A).B}$$



## Universal quantification / generalized function type constructor II

- Elimination rule ($\forall E$):

$$\frac{a : A \qquad f : (\forall x : A).B}{(f\ a) : B_{[a/x]}}$$

- Computation rule:

$$((\lambda x : A).b\ a) \to b_{[a/x]}$$



## Existential quantification / generalized product type constructor I

- Formation rule ($\exists F$)
(square brackets = discharged assumption):

$$\frac{A \text{ is a sentence/ type} \qquad \begin{matrix}[x : A] \\ \vdots \\ B \text{ is a sentence/ type}\end{matrix}}{(\exists x : A).B \text{ is a sentence/ type}}$$

- Introduction rule ($\exists I$):

$$\frac{a : A \qquad b : B_{[a/x]}}{(a, b) : (\exists x : A).B}$$

## Existential quantification / generalized product type constructor II

- Elimination rules ($\exists E_{1,2}$):

$$\frac{p : (\exists x : A).B}{Fst\ \ p : A} \qquad \frac{p : (\exists x : A).B}{Snd\ \ p : B_{[Fst\ \ p/x]}}$$

- Computation rules:

$$Fst\ \ (a, b) \rightarrow a$$
$$Snd\ \ (a, b) \rightarrow b$$



## Example proofs

- $(\forall x : A).(B \Rightarrow C) \Rightarrow (\forall x : A).B \Rightarrow (\forall x : A).C$
- $(\exists x : X).\neg P \Rightarrow \neg(\forall x : X).P$ (converse?)
- $(\exists y : Y).(\forall x : X).P \Rightarrow (\forall x : X).(\exists y : Y).P$ (converse?)