

Type Systems and Functional Programming

S.I. dr. ing. Mihnea Muraru
mmihnea@gmail.com

Computer Science Department

Fall 2017

1 / 210

Part I Introduction

2 / 210

Contents

- 1 Objectives
- 2 Functional programming

3 / 210

Contents

- 1 Objectives
- 2 Functional programming

4 / 210

Grading

- Lab: 60, ≥ 30
- Exam: 40, ≥ 20
- Final grade ≥ 50

5/210

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 . . .

6/210

One of the lab outcomes

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

7/210

Contents

- 1 Objectives
- 2 Functional programming

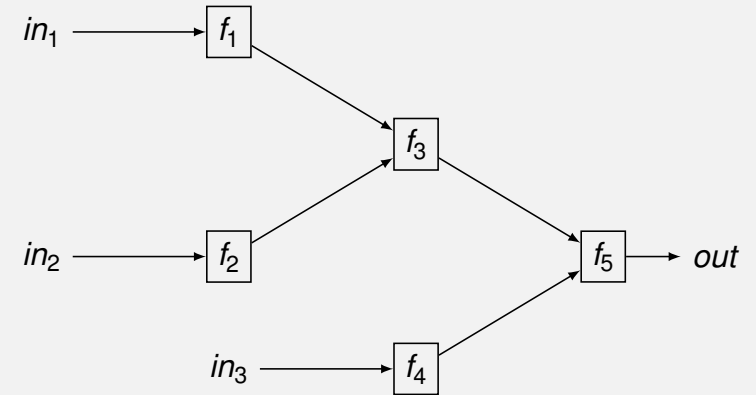
8/210

Functional programming features

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

9 / 210

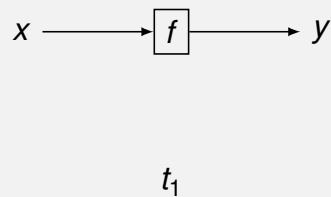
Functional flow



10 / 210

Stateless computation

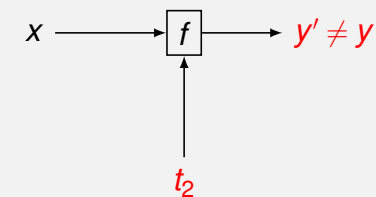
Output dependent on **input** exclusively:



11 / 210

Stateful computation

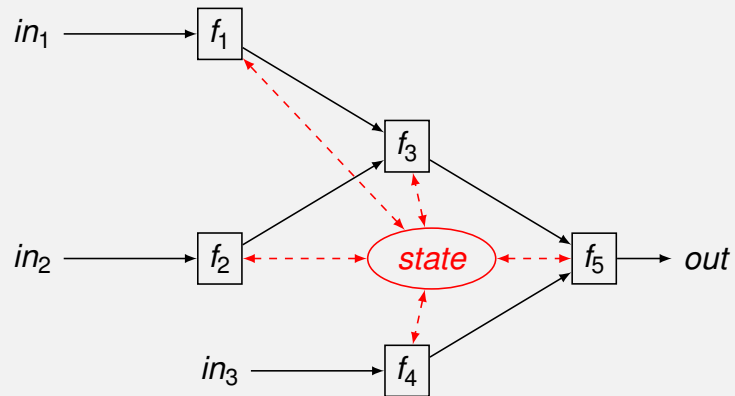
Output dependent on input and **time**:



12 / 210

Functional flow

Pure



13 / 210

Functional programming features

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

14 / 210

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

15 / 210

Part II

Untyped Lambda Calculus

16 / 210

Contents

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

17 / 210

Contents

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

18 / 210

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

19 / 210

Applications

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

20 / 210

Contents

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

21 / 210

λ -expressions

Definition

Definition 4.1 (λ -expression).

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

22 / 210

λ -expressions

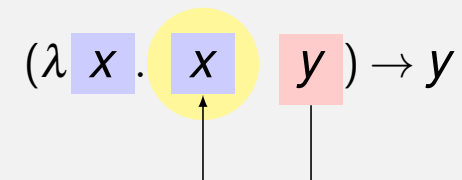
Examples

Example 4.2 (λ -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
- $(\lambda x.(x x) \lambda x.x)$

23 / 210

Intuition on application evaluation



24 / 210

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!**

25 / 210

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

26 / 210

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

27 / 210

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!**

28 / 210

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

29 / 210

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

30 / 210

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

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

31 / 210

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 λ -expressions, as in $\lambda x.((+ x) 1)$.
- Before evaluation, an expression must be brought to the **closed** form.
- The substitution process must **terminate**.

32 / 210

Contents

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

33 / 210

β -reduction

Definitions

Definition 5.1 (β -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 (β -redex).

The application $(\lambda x.E A)$.

34 / 210

β -reduction

Examples

Example 5.3 (β -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!

35 / 210

β -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).

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

36 / 210

α -conversion

Definition

Definition 5.5 (α -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 (α -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 $E_{[y/x]}$

37 / 210

α -conversion

Examples

Example 5.7 (α -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!

38 / 210

Reduction

Definitions

Definition 5.8 (Reduction step).

A sequence made of a possible α -conversion, followed by a β -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 .

39 / 210

Reduction

Examples

Example 5.10 (Reduction).

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

40 / 210

Reduction

Properties

- Reduction step = reduction sequence:

$$E_1 \rightarrow E_2 \Rightarrow E_1 \rightarrow^* E_2$$

- Reflexivity:

$$E \rightarrow^* E$$

- Transitivity:

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

41 / 210

Contents

3 Introduction

4 Lambda expressions

5 Reduction

6 Normal forms

7 Evaluation order

42 / 210

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

43 / 210

Normal forms

Definition 6.1 (Normal form).

The form of an expression that **cannot** be reduced i.e., that contains no β -redexes.

Definition 6.2 (Functional normal form, FNF).

$\lambda x.E$, **even** if E contains β -redexes.

Example 6.3 (Normal forms).

$(\lambda x.\lambda y.(x\ y)\ \lambda x.x) \rightarrow_{\text{FNF}} \lambda y.(\lambda x.x\ y) \rightarrow_{\text{NF}} \lambda y.y$

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

44 / 210

Reduction termination (reducibility)

Example 6.4.

$\Omega \equiv (\lambda x.(x\ x)\ \lambda x.(x\ x)) \rightarrow (\lambda x.(x\ x)\ \lambda x.(x\ x)) \rightarrow^* \dots$
 Ω does **not** have a terminating reduction sequence.

Definition 6.5 (Reducible expression).

An expression that has a **terminating** reduction sequence.

Ω is irreducible.

45 / 210

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

46 / 210

Reduction sequences

Example 6.6 (Reduction sequences).

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

- $\xrightarrow{1} y$
- $\xrightarrow{2} E \xrightarrow{1} y$
- $\xrightarrow{2} E \xrightarrow{2} E \xrightarrow{1} y$
- ...
- $\xrightarrow{2^n 1^*} y, n \geq 0$
- $\xrightarrow{2^\infty^*} \dots$
- E has a **nonterminating** reduction sequence, but still has a **normal form**, y . E is reducible, Ω is not.
- The length of terminating reduction sequences is **unbounded**.

47 / 210

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

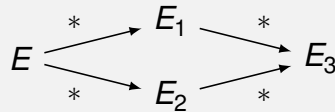
48 / 210

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.

49 / 210

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

50 / 210

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.

51 / 210

Computational equivalence

Definition 6.12 (Computational equivalence).

Two expressions are computationally equivalent iff they 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)

52 / 210

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

53 / 210

Reduction order

Definitions and examples

Definition 6.14 (Left-to-right reduction step).

The reduction of the **outermost leftmost** β -redex.

Example 6.15 (Left-to-right reduction).

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

Definition 6.16 (Right-to-left reduction step).

The reduction of the **innermost rightmost** β -redex.

Example 6.17 (Right-to-left reduction).

$((\lambda x.x \lambda x.y) (\lambda x.(x x) \lambda x.(x x))) \rightarrow (\lambda x.y \ \Omega) \rightarrow \dots$

54 / 210

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!

55 / 210

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

56 / 210

Contents

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

57 / 210

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.

58 / 210

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
```

59 / 210

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.

60 / 210

Summary

- Lambda calculus: model of computation, underpinned by functions and textual substitution
- Bound/free variables and variable occurrences w.r.t. an expression
- β -reduction, α -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!

61 / 210

Part III

Lambda Calculus as a Programming Language

62 / 210

Contents

- 8 The λ_0 language
- 9 Abstract data types (ADTs)
- 10 Implementation
- 11 Recursion
- 12 Language specification

63 / 210

Contents

- 8 The λ_0 language
- 9 Abstract data types (ADTs)
- 10 Implementation
- 11 Recursion
- 12 Language specification

64 / 210

Purpose

- Proving the **expressive** power of lambda calculus
- Hypothetical **λ -machine**
- Machine code: λ -expressions — the **λ_0 language**
- Instead of
 - bits
 - bit operations,we have
 - structured **strings** of symbols
 - **reduction** — textual substitution

65/210

λ_0 features

- Instructions:
 - **λ -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!

66/210

Shorthands

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

67/210

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**

68/210

No types

How are objects represented?

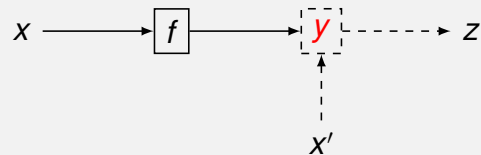
- A number, list or tree potentially designated by the **same** value e.g.,

number 3 $\rightarrow \lambda x.\lambda y.x \leftarrow \text{list } (()) (()) (())$

- Both values and operators represented by functions — **context-dependent** meaning

number 3 $\rightarrow \lambda x.\lambda y.x \leftarrow \text{operator } \textit{car}$

- **Value** applicable onto another value, as an **operator**!



69 / 210

No types

How is correctness affected?

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

70 / 210

No types

Consequences

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

71 / 210

So...

- How do we employ the λ_0 language in everyday **programming**?
- How do we represent usual **values** — numbers, booleans, lists, etc. — and their corresponding **operators**?

72 / 210

Contents

- 8 The λ_0 language
- 9 Abstract data types (ADTs)
- 10 Implementation
- 11 Recursion
- 12 Language specification

73 / 210

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, ... λ -expression!

Components:

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

74 / 210

The *Natural* ADT

Base constructors and operators

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

75 / 210

The *Natural* ADT

Axioms

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

76 / 210

Providing axioms

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

77/210

From ADTs to functional programming

Example

- **Axiome:**

- $add(zero, n) = n$
- $add(succ(m), n) = succ(add(m, n))$

- **Scheme:**

```
1 (define add
2   (lambda (m n)
3     (if (zero? m) n
4         (+ 1 (add (- m 1) n))))))
```

- **Haskell:**

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

78/210

From ADTs to functional programming

Discussion

- Proving ADT **correctness**
— structural induction
- Proving properties of **λ -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

79/210

Contents

- 8 The λ_0 language
- 9 Abstract data types (ADTs)
- 10 Implementation**
- 11 Recursion
- 12 Language specification

80/210

The *Bool* ADT

Base constructors and operators

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

81/210

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*
 - $(if\ T\ a\ b) = a$
 - $(if\ F\ a\ b) = b$

82/210

The *Bool* ADT

Base constructor implementation

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

83/210

The *Bool* ADT

Operator implementation

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

84/210

The *Pair* ADT

Specification

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

85/210

The *Pair* ADT

Implementation

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

86/210

The *List* ADT

Base constructors and operators

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

87/210

The *List* ADT

Axioms

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

88/210

The *List* ADT

Implementation

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

89 / 210

The *Natural* ADT

Axioms

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

90 / 210

The *Natural* ADT

Implementation

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

91 / 210

Contents

- 8 The λ_0 language
- 9 Abstract data types (ADTs)
- 10 Implementation
- 11 Recursion**
- 12 Language specification

92 / 210

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.,
 $id = \{(0, 0), (1, 1), (2, 2), \dots\}$

93 / 210

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

94 / 210

Implementing *length*

Problem

- Length of a list:
 $length \equiv_{\text{def}} \lambda L. (if (null? L) zero (succ (length (cdr L))))$
- What do we **replace** the underlined area with, to avoid textual recursion?
- Rewrite the definition as a **fixed-point** equation
 $Length \equiv_{\text{def}} \lambda f L. (if (null? L) zero (succ (f (cdr L))))$
 $(Length length) \rightarrow length$
- How do we **compute** the fixed point? (see code archive)

95 / 210

Contents

- 8 The λ_0 language
- 9 Abstract data types (ADTs)
- 10 Implementation
- 11 Recursion
- 12 Language specification

96 / 210

Axiomatization benefits

- Disambiguation
- Proof of properties
- Implementation skeleton

97 / 210

Syntax

- Variable:

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

- Expression:

$Expr ::= Var$
| $\lambda Var.Expr$
| $(Expr Expr)$

- Value:

$Val ::= \lambda Var.Expr$

98 / 210

Evaluation rules

Rule name:

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

99 / 210

Semantics for normal-order evaluation

Evaluation

- Reduce:

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

- Eval:

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

100 / 210

Semantics for normal-order evaluation

Substitution

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

101/210

Semantics for normal-order evaluation

Free variables

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

102/210

Semantics for normal-order evaluation

Example

Example 12.1 (Evaluation rules).

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

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

$(\lambda y. y b) \rightarrow b \quad (\text{Reduce})$

103/210

Semantics for applicative-order evaluation

Evaluation

- *Reduce* ($v \in \text{Val}$):

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

- *Eval*₁:

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

- *Eval*₂ ($v \in \text{Val}$):

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

104/210

Formal proof

Proposition 12.2 (Free and bound variables).

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

Proof.

Structural induction, according to the different forms of λ -expressions (see the lecture notes). \square

105 / 210

Summary

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

106 / 210

Part IV

Typed Lambda Calculus

107 / 210

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_ω)

108 / 210

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_ω)

109/210

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., $(x\ x)$, Ω , Fix
- Potential **irreducibility** of expressions — inconsistent representation of equivalent values

$\lambda x.(Fix\ x) \rightarrow \lambda x.(x\ (Fix\ x)) \rightarrow \lambda x.(x\ (x\ (Fix\ x))) \rightarrow \dots$

110/210

Solution

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

111/210

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**.

112/210

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_ω)

113/210

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., $\sigma \rightarrow \tau$, where σ and τ are types.

Notation:

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

114/210

Typed λ -expressions

Definition 14.3 (λ_t -expression).

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

115/210

Relation to untyped lambda calculus

Similarities

- β -reduction
- α -conversion
- normal forms
- Church-Rosser theorem

Differences

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

116/210

Syntax

Expressions

- Variables:

$$Var ::= \dots$$

- Expressions:

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

- Values:

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

117/210

Syntax

Types

- Types:

$$\begin{aligned} Type & ::= BaseType \\ & | (Type \rightarrow Type) \end{aligned}$$

- Typing contexts:

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

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

118/210

Semantics for normal-order evaluation

Evaluation

- Reduce:

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

- Eval:

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

The type annotations are **ignored**,
since typing **precedes** evaluation.

119/210

Semantics

Typing

- TBaseVal:

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

- TVar:

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

- TAbs:

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

- TApp:

$$\frac{\Gamma \vdash e : (\tau' \rightarrow \tau) \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e \ e') : \tau}$$

120/210

Typing example

Example 14.4 (Typing).

$$\lambda X : \tau_1. \lambda Y : \tau_2. X : (\tau_1 \rightarrow (\tau_2 \rightarrow \tau_1))$$

Blackboard!

121 / 210

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.

122 / 210

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_ω)

123 / 210

Ways of extending STLC

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

124 / 210

The product type

Algebraic specification

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

125 / 210

The product type

Syntax

$$\begin{aligned} \text{Expr} & ::= \dots \\ & | \text{fst Expr} \\ & | \text{snd Expr} \\ & | (\text{Expr}, \text{Expr}) \end{aligned}$$
$$\begin{aligned} \text{BaseVal} & ::= \dots \\ & | \text{ProductVal} \end{aligned}$$
$$\text{ProductVal} ::= (\text{Val}, \text{Val})$$
$$\begin{aligned} \text{Type} & ::= \dots \\ & | (\text{Type} * \text{Type}) \end{aligned}$$

126 / 210

The product type

Evaluation

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

127 / 210

The product type

Typing

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

128 / 210

The product type

Typing example

Example 15.1 (Typing).

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

Blackboard!

129 / 210

The *Bool* type

Algebraic specification

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

130 / 210

The *Bool* type

Syntax

$$Expr ::= \dots$$
$$| (if Expr Expr Expr)$$
$$BaseVal ::= \dots$$
$$| BoolVal$$
$$BoolVal ::= True \mid False$$
$$BaseType ::= \dots$$
$$| Bool$$

131 / 210

The *Bool* type

Evaluation

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

132 / 210

The *Bool* type

Typing

- *TTrue*:

$$\Gamma \vdash \text{True} : \text{Bool}$$

- *TFalse*:

$$\Gamma \vdash \text{False} : \text{Bool}$$

- *TIf*:

$$\frac{\Gamma \vdash e : \text{Bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (\text{if } e \ e_1 \ e_2) : \tau}$$

133 / 210

The *Bool* type

Top-level variable bindings

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

134 / 210

The \mathbb{N} type

Algebraic specification

- Base constructors i.e., canonical values:

- $\mathbb{N} ::= 0 \mid (\text{succ } \mathbb{N})$

- Operators:

- $+$: $\mathbb{N}^2 \rightarrow \mathbb{N}$
- zero? : $\mathbb{N} \rightarrow \text{Bool}$

- Axioms ($m, n \in \mathbb{N}$):

- $(+ \ 0 \ n) = n$
- $(+ \ (\text{succ } m) \ n) = (\text{succ } (+ \ m \ n))$
- $(\text{zero? } 0) = \text{True}$
- $(\text{zero? } (\text{succ } n)) = \text{False}$

135 / 210

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} , $\text{prec}_{\mathbb{N}}$, which allows for defining any primitive recursive function on natural numbers
- Define the **operators** using the primitive recursor

136 / 210

The \mathbb{N} type

Syntax

$$\begin{aligned} \text{Expr} & ::= \dots \\ & | (\text{succ Expr}) \\ & | (\text{prec}_{\mathbb{N}} \text{Expr Expr Expr}) \end{aligned}$$
$$\begin{aligned} \text{BaseVal} & ::= \dots \\ & | \text{NVal} \end{aligned}$$
$$\begin{aligned} \text{NVal} & ::= 0 \\ & | (\text{succ NVal}) \end{aligned}$$
$$\begin{aligned} \text{BaseType} & ::= \dots \\ & | \mathbb{N} \end{aligned}$$

137 / 210

The \mathbb{N} type

Evaluation

- *EvalSucc*:

$$\frac{e \rightarrow e'}{(\text{succ } e) \rightarrow (\text{succ } e')}$$

- *EvalPrec_{N0}*:

$$(\text{prec}_{\mathbb{N}} e_0 f 0) \rightarrow e_0$$

- *EvalPrec_{N1}* ($n \in \mathbb{N}$):

$$(\text{prec}_{\mathbb{N}} e_0 f (\text{succ } n)) \rightarrow (f n (\text{prec}_{\mathbb{N}} e_0 f n))$$

- *EvalPrec_{N2}*:

$$\frac{e \rightarrow e'}{(\text{prec}_{\mathbb{N}} e_0 f e) \rightarrow (\text{prec}_{\mathbb{N}} e_0 f e')}$$

138 / 210

The \mathbb{N} type

Typing

- *TZero*:

$$\Gamma \vdash 0 : \mathbb{N}$$

- *TSucc*:

$$\frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash (\text{succ } e) : \mathbb{N}}$$

- *TPrec_N*:

$$\frac{\Gamma \vdash e_0 : \tau \quad \Gamma \vdash f : \mathbb{N} \rightarrow \tau \rightarrow \tau \quad \Gamma \vdash e : \mathbb{N}}{\Gamma \vdash (\text{prec}_{\mathbb{N}} e_0 f e) : \tau}$$

139 / 210

The \mathbb{N} type

Top-level variable bindings

- $\text{zero?} \equiv \lambda n : \mathbb{N}. (\text{prec}_{\mathbb{N}} \text{True } \lambda x : \mathbb{N}. \lambda y : \text{Bool}. \text{False } n)$

- $+$ $\equiv \lambda m : \mathbb{N}. \lambda n : \mathbb{N}. (\text{prec}_{\mathbb{N}} n \lambda x : \mathbb{N}. \lambda y : \mathbb{N}. (\text{succ } y) m)$

140 / 210

The (*List* τ) type

Algebraic specification

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

141 / 210

The (*List* τ) type

Syntax

$$\begin{aligned} \text{Expr} & ::= \dots \\ & | (\text{cons Expr Expr}) \\ & | (\text{prec}_L \text{Expr Expr Expr}) \end{aligned}$$
$$\begin{aligned} \text{BaseVal} & ::= \dots \\ & | \text{ListVal} \end{aligned}$$
$$\begin{aligned} \text{ListVal} & ::= [] \\ & | (\text{cons Value ListVal}) \end{aligned}$$
$$\begin{aligned} \text{Type} & ::= \dots \\ & | (\text{List Type}) \end{aligned}$$

142 / 210

The (*List* τ) type

Evaluation

- *EvalCons*:

$$\frac{e \rightarrow e'}{(\text{cons } e \ e'') \rightarrow (\text{cons } e' \ e'')}$$

- *EvalPrec_{L0}*:

$$(\text{prec}_L \ e_0 \ f \ []) \rightarrow e_0$$

- *EvalPrec_{L1}* ($v \in \text{Value}$):

$$(\text{prec}_L \ e_0 \ f \ (\text{cons } v \ e)) \rightarrow (f \ v \ e \ (\text{prec}_L \ e_0 \ f \ e))$$

- *EvalPrec_{L2}*:

$$\frac{e \rightarrow e'}{(\text{prec}_L \ e_0 \ f \ e) \rightarrow (\text{prec}_L \ e_0 \ f \ e')}$$

143 / 210

The (*List* τ) type

Typing

- *TEmpty*:

$$\Gamma \vdash []_{\tau} : (\text{List } \tau)$$

- *TCons*:

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : (\text{List } \tau)}{\Gamma \vdash (\text{cons } e \ e') : (\text{List } \tau)}$$

- *TPrec_L*:

$$\frac{\Gamma \vdash e_0 : \tau' \quad \Gamma \vdash f : \tau \rightarrow (\text{List } \tau) \rightarrow \tau' \rightarrow \tau' \quad \Gamma \vdash e : (\text{List } \tau)}{\Gamma \vdash (\text{prec}_L \ e_0 \ f \ e) : \tau'}$$

144 / 210

The (*List* τ) type

Top-level variable bindings

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

145 / 210

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

146 / 210

fix

Syntax

$$\begin{array}{l} Expr ::= \dots \\ | \quad (fix\ Expr) \end{array}$$

147 / 210

fix

Evaluation

- *EvalFix*:

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

- *EvalFix'*:

$$\frac{e \rightarrow e'}{(fix\ e) \rightarrow (fix\ e')}$$

148 / 210

- *TFix*:

$$\frac{\Gamma \vdash e : (\tau \rightarrow \tau)}{\Gamma \vdash (\text{fix } e) : \tau}$$

Example 15.2 (The *remainder* function).

remainder = $\lambda m : \mathbb{N} . \lambda n : \mathbb{N} .$

$((\text{fix } \lambda f : (\mathbb{N} \rightarrow \mathbb{N}) . \lambda p : \mathbb{N} .$

$(\text{if } p < n \text{ then } p \text{ else } (f (p - n)))) m$

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

Monomorphism

- Within the types $(\tau * \tau')$ and $(\text{List } \tau)$, τ and τ' designate **specific** types e.g., *Bool*, \mathbb{N} , $(\text{List } \mathbb{N})$, etc.
- **Dedicated** operators for each simple type
- $\text{fst}_{\mathbb{N}, \text{Bool}}$, $\text{fst}_{\text{Bool}, \mathbb{N}}$, ...
- $\llbracket \cdot \rrbracket_{\mathbb{N}}$, $\llbracket \cdot \rrbracket_{\text{Bool}}$, ...
- $\text{empty?}_{\mathbb{N}}$, $\text{empty?}_{\text{Bool}}$, ...

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_ω)

Idea

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

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

- **Polymorphic** identity function — type variables:

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

- **Type coercion** prior to function application:

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

153 / 210

Syntax

- **Program** variables: stand for program values

$$Var ::= \dots$$

- **Type** variables: stand for types

$$TypeVar ::= \dots$$

154 / 210

Syntax

- Expressions:

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

- Values:

$$\begin{array}{l} Value ::= BaseValue \\ | \lambda Var : Type. Expr \\ | \lambda TypeVar. Expr \end{array}$$

155 / 210

Syntax

- Types:

$$\begin{array}{l} Type ::= BaseType \\ | TypeVar \\ | (Type \rightarrow Type) \\ | \forall TypeVar. Type \end{array}$$

- Typing contexts:

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

156 / 210

Semantics

Evaluation

- *Reduce*₁:

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

- *Reduce*₂:

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

- *Eval*₁:

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

- *Eval*₂:

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

157 / 210

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:

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

- *TApp*₁:

$$\frac{\Gamma \vdash e : (\tau' \rightarrow \tau) \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e \ e') : \tau}$$

158 / 210

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:

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

159 / 210

Semantics

Substitution and free variables

- *Expr*_[Expr/Var]

- *Expr*_[Type/TypeVar]

- *Type*_[Type/TypeVar]

- Free program variables

- Free type variables

160 / 210

Typing example

Example 16.1 (Typing).

$$\begin{aligned} \Gamma \vdash \lambda f : \forall X. (X \rightarrow X). \lambda Y. \lambda x : Y. (f[Y] x) \\ : (\forall X. (X \rightarrow X) \rightarrow \forall Y. (Y \rightarrow Y)) \end{aligned}$$

Monomorphic function
with polymorphic argument and result!

Blackboard!

161 / 210

Examples of polymorphic expressions

Example 16.2 (Doubling a computation).

$$\begin{aligned} \text{double} &\equiv \lambda X. \lambda f : (X \rightarrow X). \lambda x : X. (f (f x)) \\ &: \forall X. ((X \rightarrow X) \rightarrow (X \rightarrow X)) \end{aligned}$$

Example 16.3 (Quadrupling a computation).

$$\begin{aligned} \text{quadruple} &\equiv \lambda X. (\text{double}[X \rightarrow X] \text{ double}[X]) \\ &: \forall X. ((X \rightarrow X) \rightarrow (X \rightarrow X)) \end{aligned}$$

162 / 210

Examples of polymorphic expressions

Example 16.4 (Reflexive computation).

$$\begin{aligned} \text{reflexive} &\equiv \lambda f : \forall X. (X \rightarrow X). (f[\forall X. (X \rightarrow X)] f) \\ &: (\forall X. (X \rightarrow X) \rightarrow \forall X. (X \rightarrow X)) \end{aligned}$$

Example 16.5 (Fixed-point combinator).

$$\begin{aligned} \text{Fix} &\equiv \lambda X. \lambda f : (X \rightarrow X). (f (\text{Fix}[X] f)) \\ &: \forall X. ((X \rightarrow X) \rightarrow X) \end{aligned}$$

163 / 210

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_ω)

164 / 210

Motivation

165 / 210

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_ω)

166 / 210

Problem

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

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

- 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 $(List\ \mathbb{N})$:

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

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

167 / 210

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**

168 / 210

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** τ : $\tau :: k$

169 / 210

Kinds

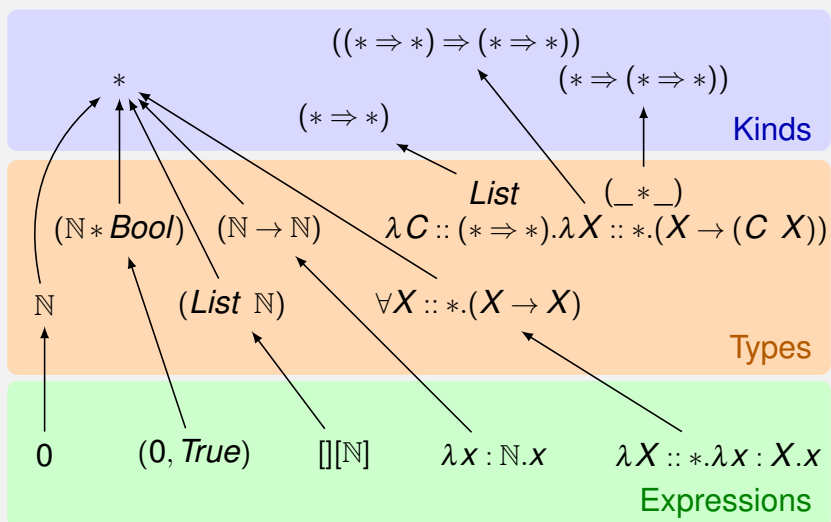
Examples

Example 18.1 (Kinds).

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

170 / 210

Levels of expressions



171 / 210

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**: $\tau_1 \equiv \tau_2$

172 / 210

Syntax

- Expressions:

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

- Values:

$$\begin{aligned} \text{Value} & ::= \text{BaseValue} \\ & | \lambda \text{Var} : \text{Type}. \text{Expr} \\ & | \lambda \text{TypeVar} :: \text{Kind}. \text{Expr} \end{aligned}$$

173 / 210

Syntax

- Types:

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

- Typing contexts:

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

174 / 210

Syntax

- Kinds:

$$\begin{aligned} \text{Kind} & ::= * \\ & | (\text{Kind} \Rightarrow \text{Kind}) \end{aligned}$$

175 / 210

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 :

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

176 / 210

Semantics

Typing

- *TBaseValue*:

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

- *TVar*:

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

- *TAbs₁*:

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

- *TApp₁*:

$$\frac{\Gamma \vdash e : (\tau' \rightarrow \tau) \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash (e e') : \tau}$$

177 / 210

Semantics

Typing

- *TAbs₂*:

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

- *TApp₂*:

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

178 / 210

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*:

$$\frac{\Gamma \vdash \tau :: (K' \Rightarrow K) \quad \Gamma \vdash \tau' :: K'}{\Gamma \vdash (\tau \tau') :: K}$$

179 / 210

Semantics

Kinding

- *KAbs₁*:

$$\frac{\Gamma \vdash \tau :: * \quad \Gamma \vdash \tau' :: *}{\Gamma \vdash (\tau \rightarrow \tau') :: *}$$

- *KAbs₂*:

$$\frac{\Gamma, X :: K \vdash \tau :: *}{\Gamma \vdash \forall X :: K. \tau :: *}$$

180 / 210

Semantics

Type equivalence

- *EqReflexivity*:

$$\tau \equiv \tau$$

- *EqSymmetry*:

$$\frac{\tau \equiv \tau'}{\tau' \equiv \tau}$$

- *EqTransitivity*:

$$\frac{\tau \equiv \tau' \quad \tau' \equiv \tau''}{\tau \equiv \tau''}$$

- *EqTypeReduce*:

$$(\lambda X :: K. \tau \ \tau') \equiv \tau_{[\tau'/X]}$$

181 / 210

Semantics

Type equivalence

- *EqTypeAbs*:

$$\frac{\tau \equiv \tau'}{\lambda X :: K. \tau \equiv \lambda X :: K. \tau'}$$

- *EqTypeApp*:

$$\frac{\tau \equiv \tau' \quad \sigma \equiv \sigma'}{(\tau \ \sigma) \equiv (\tau' \ \sigma')}$$

- *EqAbs₁*:

$$\frac{\tau \equiv \tau' \quad \sigma \equiv \sigma'}{(\tau \rightarrow \sigma) \equiv (\tau' \rightarrow \sigma')}$$

- *EqAbs₂*:

$$\frac{\tau \equiv \tau'}{\forall X :: K. \tau \equiv \forall X :: K. \tau'}$$

182 / 210

Semantics

Type equivalence

- *TypeEquivalence*:

$$\frac{\Gamma \vdash e : \tau \quad \tau \equiv \tau'}{\Gamma \vdash e : \tau'}$$

183 / 210

Kinding example

Example 18.2 (Kinding).

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

Blackboard!

184 / 210

Part V

Constructive Type Theory

185 / 210

Contents

19 Constructive paradigm

20 Syntax and semantics

186 / 210

Contents

19 Constructive paradigm

20 Syntax and semantics

187 / 210

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

188 / 210

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

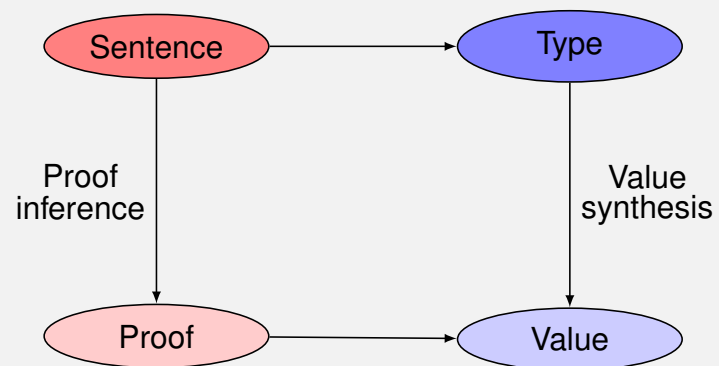
189 / 210

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

190 / 210

The Curry-Howard isomorphism



191 / 210

Contents

- 19 Constructive paradigm
- 20 Syntax and semantics

192 / 210

Two views

$a : A$

- Type-theoretic: “ a is a value of type A ”
- Logical: “ a is a proof of sentence A ”

193 / 210

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

194 / 210

Other logic-type correspondences

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

195 / 210

Logical conjunction / product type constructor I

- Formation rule ($\wedge F$):

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

- Introduction rule ($\wedge I$):

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

196 / 210

Logical conjunction / product type constructor II

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

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

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

- Computation rules:

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

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

197 / 210

Logical implication / function type constructor I

- Formation rule ($\Rightarrow F$):

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

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

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

198 / 210

Logical implication / function type constructor II

- Elimination rule ($\Rightarrow E$):

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

- Computation rule:

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

199 / 210

Logical disjunction / sum type constructor I

- Formation rule ($\vee F$):

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

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

$$\frac{a : A}{inl\ a : A \vee B}$$

$$\frac{b : B}{inr\ b : A \vee B}$$

200 / 210

Logical disjunction / sum type constructor II

- Elimination rule ($\vee E$):

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

- Computation rules:

$$\begin{aligned} \text{cases } (\text{inl } a) \ f \ g &\rightarrow f \ a \\ \text{cases } (\text{inr } b) \ f \ g &\rightarrow g \ b \end{aligned}$$

201 / 210

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

202 / 210

Absurd sentence / empty type II

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

$$\frac{p : \perp}{\text{abort}_A \ p : A}$$

- Computation rule: none

203 / 210

Logical negation and equivalence

- Logical negation:

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

- Logical equivalence:

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

204 / 210

Example proofs

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

205 / 210

Universal quantification / generalized function type constructor I

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

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

- Introduction rule ($\forall I$):

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

206 / 210

Universal quantification / generalized function type constructor II

- Elimination rule ($\forall E$):

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

- Computation rule:

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

207 / 210

Existential quantification / generalized product type constructor I

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

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

- Introduction rule ($\exists I$):

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

208 / 210

Existential quantification / generalized product type constructor II

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

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

- Computation rules:

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

209 / 210

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?)

210 / 210