Logical Foundations of Programming Languages

Olivier LAURENT & Colin RIBA

LIP - ENS de Lyon

Course 01

A Naive Introduction

A First Example

Consider

```
foo(f:int->int):
  return f((10^10)!)*0
```

A First Example

Consider

```
foo(f:int->int):
  return f((10^10)!)*0
```

and

```
bar(f:int->int):
  return 0
```

A First Example

Consider

```
foo(f:int->int):
  return f((10^10)!)*0
```

and

```
bar(f:int->int):
  return 0
```

Are these two programs equivalent ?

A First Example

Consider

```
foo(f:int->int):
  return f((10^10)!)*0
```

and

```
bar(f:int->int):
  return 0
```

Are these two programs equivalent ?

- They are **not** equivalent if `f` can throw an exception (e.g. division by 0).
- They are **not** equivalent if `f` does not terminate.
- They are **not** equivalent in terms of exection time.

A First Example

Consider

```
foo(f:int->int):
  return f((10^10)!)*0
```

and

```
bar(f:int->int):
  return 0
```

Are these two programs equivalent ?

- They are **not** equivalent if **f** can throw an exception (e.g. division by 0).
- They are **not** equivalent if **f** does not terminate.
- They are **not** equivalent in terms of exection time.
- They **are** equivalent if **f** behaves as a **function**

$$[\![\mathtt{f}]\!] \quad : \quad [\![\mathtt{int}]\!] \longrightarrow [\![\mathtt{int}]\!]$$

where $[\![\mathtt{int}]\!]$ is the set of integers $\mathbb{Z}$.

Mathematical Models of Programming Languages

```
foo(f:int->int):
  return f((10^10)!)*0
```

vs

```
bar(f:int->int):
  return 0
```

Mathematical Models of Programming Languages

```
foo(f:int->int):
  return f((10^10)!)*0
```

vs

```
bar(f:int->int):
  return 0
```

Naive Idea:

$$\begin{array}{ccc} \text{Types} & \equiv & \text{Sets} \\ \text{Programs} & \equiv & \text{Functions} \end{array}$$

Mathematical Models of Programming Languages

```
foo(f:int->int):
  return f((10^10)!)*0
```

vs

```
bar(f:int->int):
  return 0
```

Naive Idea:

Types	$\equiv$	Sets
Programs	$\equiv$	Functions

Curry-Howard-Lambek Correspondence:

Prog. Languages		Logic
Types	$\equiv$	Formulae
Programs	$\equiv$	Proofs

Logical Foundations of Programming Languages

```
foo(f:int->int):
  return f((10^10)!)*0
```

vs

```
bar(f:int->int):
  return 0
```

Naive Idea:

Types	$\equiv$	Sets
Programs	$\equiv$	Functions

Curry-Howard-Lambek Correspondence:

Prog. Languages		Logic
Types	$\equiv$	Formulae
Programs	$\equiv$	Proofs

Logical Foundations of Programming Languages

```
foo(f:int->int):
  return f((10^10)!)*0
```

vs

```
bar(f:int->int):
  return 0
```

Naive Idea:

Types	$\equiv$	Sets
Programs	$\equiv$	Functions

Curry-Howard-Lambek Correspondence:

Prog. Languages		Logic		Categories
Types	$\equiv$	Formulae	$\equiv$	Objects
Programs	$\equiv$	Proofs	$\equiv$	Morphisms

Logical Foundations of Programming Languages

```
foo(f:int->int):
  return f((10^10)!)*0
```

vs

```
bar(f:int->int):
  return 0
```

Naive Idea:

Types	$\equiv$	Sets
Programs	$\equiv$	Functions

Curry-Howard-Lambek Correspondence:

Prog. Languages		Logic		Categories
Types	$\equiv$	Formulae	$\equiv$	Objects
Programs	$\equiv$	Proofs	$\equiv$	Morphisms

- Linear Logic (O. LAURENT)

Logical Foundations of Programming Languages

```
foo(f:int->int):
  return f((10^10)!)*0
```

vs

```
bar(f:int->int):
  return 0
```

Naive Idea:

Types	$\equiv$	Sets
Programs	$\equiv$	Functions

Curry-Howard-Lambek Correspondence:

Prog. Languages		Logic		Categories
Types	$\equiv$	Formulae	$\equiv$	Objects
Programs	$\equiv$	Proofs	$\equiv$	Morphisms

- Linear Logic (O. LAURENT)

Keywords.

- Type systems, proof theory, denotational semantics, category theory.

Logical Foundations of Programming Languages

```
foo(f:int->int):
  return f((10^10)!)*0
```

vs

```
bar(f:int->int):
  return 0
```

Naive Idea:

Types	$\equiv$	Sets
Programs	$\equiv$	Functions

Curry-Howard-Lambek Correspondence:

Prog. Languages		Logic		Categories
Types	$\equiv$	Formulae	$\equiv$	Objects
Programs	$\equiv$	Proofs	$\equiv$	Morphisms

- Linear Logic (O. LAURENT)

Keywords.

- Type systems, proof theory, denotational semantics, category theory.

This part of the course.

- Mathematical models of programming languages.

Logical Foundations of Programming Languages

```
foo(f:int->int):
  return f((10^10)!)*0
```

vs

```
bar(f:int->int):
  return 0
```

Naive Idea:

Types	$\equiv$	Sets
Programs	$\equiv$	Functions

Curry-Howard-Lambek Correspondence:

Prog. Languages		Logic		Categories
Types	$\equiv$	Formulae	$\equiv$	Objects
Programs	$\equiv$	Proofs	$\equiv$	Morphisms

- Linear Logic (O. LAURENT)

Keywords.

- Type systems, proof theory, denotational semantics, category theory.

This part of the course.

- Mathematical models of programming languages.

Now:

- A naive introduction to some basic ideas.

Types as Sets

- Assume types, say $\mathbf{\texttt{int}}$, $\mathbf{\texttt{bool}}$, are to be interpreted as sets $[\![\mathbf{\texttt{int}}]\!]$, $[\![\mathbf{\texttt{bool}}]\!]$.

Types as Sets

- Assume types, say $\mathtt{int}$, $\mathtt{bool}$, are to be interpreted as sets $[\![\mathtt{int}]\!]$, $[\![\mathtt{bool}]\!]$.

Question.

- Can we assume

$$[\![\mathtt{bool}]\!] \quad := \quad \{\mathtt{true}, \mathtt{false}\} \tag{1}$$

Types as Sets

- Assume types, say **int**, **bool**, are to be interpreted as sets $[\![\mathtt{int}]\!]$, $[\![\mathtt{bool}]\!]$.

Question.

- Can we assume

$$[\![\mathtt{bool}]\!] \quad := \quad \{\mathtt{true}, \mathtt{false}\} \tag{1}$$

Answer.

- Consider the non-terminating program

```
loop (b:bool):
  while true:
    skip
  return true
```

Types as Sets

- Assume types, say **int**, **bool**, are to be interpreted as sets $[\![\mathtt{int}]\!]$, $[\![\mathtt{bool}]\!]$.

Question.

- Can we assume

$$[\![\mathtt{bool}]\!] \quad := \quad \{\mathtt{true}, \mathtt{false}\} \tag{1}$$

Answer.

- Consider the non-terminating program

```
loop (b:bool):
  while true:
    skip
  return true
```

- If $[\![\mathtt{bool}]\!]$ is as in (1), then we cannot have

$$[\![\mathtt{loop}]\!] \quad : \quad [\![\mathtt{bool}]\!] \quad \longrightarrow \quad [\![\mathtt{bool}]\!]$$

Types as Sets

- Assume types, say **int**, **bool**, are to be interpreted as sets $[\![\texttt{int}]\!]$, $[\![\texttt{bool}]\!]$.

Question.

- Can we assume

$$[\![\texttt{bool}]\!] \quad := \quad \{\texttt{true}, \texttt{false}\} \tag{1}$$

Answer.

- Consider the non-terminating program

```
loop (b:bool):
  while true:
    skip
  return true
```

- If $[\![\texttt{bool}]\!]$ is as in (1), then we cannot have

$$[\![\texttt{loop}]\!] \quad : \quad [\![\texttt{bool}]\!] \quad \longrightarrow \quad [\![\texttt{bool}]\!]$$

- We shall therefore represent divergence and assume

$$[\![\texttt{bool}]\!] \quad := \quad \{\bot, \texttt{true}, \texttt{false}\} \qquad \text{($\bot$ “$=$” divergence)}$$

We can then have, as expected:

$$[\![\texttt{loop}]\!](a) \quad = \quad \bot \qquad \text{(for all $a \in [\![\texttt{bool}]\!]$)}$$

A Taste of Finitary PCF

Motivation.

- A simple language to discuss

$$[\![\mathtt{bool}]\!] \quad := \quad \{\bot, \mathtt{true}, \mathtt{false}\}$$

A Taste of Finitary PCF

Motivation.

- A simple language to discuss

$$[\![\mathtt{bool}]\!] \quad := \quad \{\perp, \mathtt{true}, \mathtt{false}\}$$

The Language of Finitary PCF.

$$\begin{array}{lcl} \tau, \sigma & ::= & \mathtt{bool} \mid \sigma \to \tau \\ t, u & ::= & t\,u \mid \mathtt{fun}\, x \to t \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\ t\ \mathtt{then}\ u\ \mathtt{else}\ v \mid \Omega \end{array}$$

A Taste of Finitary PCF

Motivation.

- A simple language to discuss

$$[\![\mathtt{bool}]\!] \quad := \quad \{\bot, \mathtt{true}, \mathtt{false}\}$$

The Language of Finitary PCF.

$$\begin{array}{lcl} \tau, \sigma & ::= & \mathtt{bool} \mid \sigma \to \tau \\ t, u & ::= & t\,u \mid \mathtt{fun}\, x \to t \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\, t\, \mathtt{then}\, u\, \mathtt{else}\, v \mid \Omega \end{array}$$

- Purely functional language with Booleans and divergence (Ω).

A Taste of Finitary PCF

Motivation.

- A simple language to discuss

$$[\![\mathtt{bool}]\!] \quad := \quad \{\bot, \mathtt{true}, \mathtt{false}\}$$

The Language of Finitary PCF.

$$\begin{array}{lcl} \tau, \sigma & ::= & \mathtt{bool} \mid \sigma \to \tau \\ t, u & ::= & t\,u \mid \mathtt{fun}\,x \to t \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\,t\,\mathtt{then}\,u\,\mathtt{else}\,v \mid \Omega \end{array}$$

- Purely functional language with Booleans and divergence (Ω).

We assume *call-by-name* evaluation:

$$\begin{array}{rcl} (\mathtt{fun}\,x \to t)u & = & t[u/x] \\ \mathtt{if}\,\mathtt{true}\,\mathtt{then}\,t\,\mathtt{else}\,u & = & t \\ \mathtt{if}\,\mathtt{false}\,\mathtt{then}\,t\,\mathtt{else}\,u & = & u \end{array}$$

Example.

Consider the following two `or` programs:

```
or_l := fun a, b ->
  if a then a else b
```

vs

```
or_r := fun a, b ->
  if b then b else a
```

Example.

Consider the following two `or` programs:

```
or_l := fun a, b ->
  if a then a else b
```

vs

```
or_r := fun a, b ->
  if b then b else a
```

Questions.

- What are the functions $[\![\texttt{or\_l}]\!]$, $[\![\texttt{or\_r}]\!]$?
- Are the programs `or_l` and `or_r` equivalent ?

Example.
Consider the following two **or** programs:

```
or_l := fun a, b ->
  if a then a else b
```

vs

```
or_r := fun a, b ->
  if b then b else a
```

Questions.

- What are the functions $[\![\texttt{or\_l}]\!]$, $[\![\texttt{or\_r}]\!]$?
- Are the programs **or_l** and **or_r** equivalent ?

Example.
Consider, for $\texttt{b} \in \{\texttt{true}, \texttt{false}\}$, the program

```
taste_b := fun f ->
  if f(true, Ω) and
     f(Ω, true) and
     not(f(false, false))
  then b
  else true
```

Example.
Consider the following two **or** programs:

```
or_l := fun a, b ->
  if a then a else b
```

vs

```
or_r := fun a, b ->
  if b then b else a
```

Questions.

- What are the functions $[\![$**or_l**$]\!]$, $[\![$**or_r**$]\!]$?
- Are the programs **or_l** and **or_r** equivalent ?

Example.
Consider, for **b** $\in$ {**true**, **false**}, the program

```
taste_b := fun f ->
  if f(true, Ω) and
     f(Ω, true) and
     not(f(false, false))
  then b
  else true
```

Questions.

- Do we have $[\![$**taste_true**$]\!] = [\![$**taste_false**$]\!]$?
- Are **taste_true** and **taste_false** equivalent ?

A Taste of PCF

Motivation.

- Extend Finitary PCF with general recursion (= fixpoint combinator).
- Mathematically cleaner if an infinite type is assumed (say the natural numbers).

A Taste of PCF

Motivation.

- Extend Finitary PCF with general recursion (= fixpoint combinator).
- Mathematically cleaner if an infinite type is assumed (say the natural numbers).

The Language of PCF.

$$\begin{array}{rclcl} \tau, \sigma & ::= & \dots & \mid & \mathbf{nat} \\ t, u & ::= & \dots & \mid & t\mathtt{+1} \mid t\mathtt{-1} \mid \mathtt{z?} \mid Y \mid \underline{n} \qquad \text{(for each } n \in \mathbb{N}) \end{array}$$

- Y is a *fixpoint* combinator:

$$Y\,t \quad = \quad t\,(Y\,t)$$

A Taste of PCF

Motivation.

- Extend Finitary PCF with general recursion (= fixpoint combinator).
- Mathematically cleaner if an infinite type is assumed (say the natural numbers).

The Language of PCF.

$$\begin{array}{rclllllllllll} \tau, \sigma & ::= & \ldots & \mid & \mathbf{nat} \\ t, u & ::= & \ldots & \mid & t\mathtt{+1} & \mid & t\mathtt{-1} & \mid & \mathtt{z?} & \mid & Y & \mid & \underline{n} \end{array} \qquad \text{(for each } n \in \mathbb{N}\text{)}$$

- Y is a *fixpoint* combinator:

$$Y\,t \quad = \quad t\,(Y\,t)$$

Examples.

- We could have defined

$$\Omega \quad := \quad Y(\mathtt{fun}\,x \to x)$$

A Taste of PCF

Motivation.

- Extend Finitary PCF with general recursion (= fixpoint combinator).
- Mathematically cleaner if an infinite type is assumed (say the natural numbers).

The Language of PCF.

$$\begin{array}{rcl} \tau, \sigma & ::= & \ldots \mid \mathbf{nat} \\ t, u & ::= & \ldots \mid t\mathtt{+1} \mid t\mathtt{-1} \mid \mathtt{z?} \mid Y \mid \underline{n} \qquad \text{(for each } n \in \mathbb{N}\text{)} \end{array}$$

- Y is a *fixpoint* combinator:

$$Y\,t \quad = \quad t\,(Y\,t)$$

Examples.

- We could have defined

$$\Omega \quad := \quad Y(\mathtt{fun}\,x \to x)$$

- Addition

$$\begin{array}{rcl} \mathtt{add}\ \underline{0}\ u & = & u \\ \mathtt{add}\ t\mathtt{+1}\ u & = & (\mathtt{add}\ t\ u)\mathtt{+1} \end{array}$$

A Taste of PCF

Motivation.

- Extend Finitary PCF with general recursion (= fixpoint combinator).
- Mathematically cleaner if an infinite type is assumed (say the natural numbers).

The Language of PCF.

$$\begin{array}{lcl} \tau, \sigma & ::= & \dots \mid \mathbf{nat} \\ t, u & ::= & \dots \mid t\mathtt{+1} \mid t\mathtt{-1} \mid \mathtt{z?} \mid Y \mid \underline{n} \qquad \text{(for each } n \in \mathbb{N}\text{)} \end{array}$$

- Y is a *fixpoint* combinator:

$$Y\, t \quad = \quad t\,(Y\, t)$$

Examples.

- We could have defined

$$\Omega \quad := \quad Y(\mathbf{fun}\, x \to x)$$

- Addition

$$\begin{array}{lcl} \mathtt{add}\ \underline{0}\ u & = & u \\ \mathtt{add}\ t\mathtt{+1}\ u & = & (\mathtt{add}\ t\ u)\mathtt{+1} \end{array}$$

can be defined as

$$\mathtt{add} \quad := \quad Y\, \mathtt{add\_rec}$$

where

```
add_rec := fun f, x, y ->
  if (z? x) then y else (f x-1 y)+1
```

A Denotational Semantics for PCF ?

A Denotational Semantics for PCF ?

We would like

- each type τ to be interpreted as a set $[\![\tau]\!]$,

A Denotational Semantics for PCF ?

We would like

- each type τ to be interpreted as a set $[\![\tau]\!]$,
- a program *t* of type say $\sigma \to \tau$ to be interpreted as a function

$$[\![t]\!] \quad : \quad [\![\sigma]\!] \quad \longrightarrow \quad [\![\tau]\!]$$

A Denotational Semantics for PCF ?

We would like

- each type τ to be interpreted as a set $[\![\tau]\!]$,
- a program t of type say $\sigma \to \tau$ to be interpreted as a function

$$[\![t]\!] \quad : \quad [\![\sigma]\!] \quad \longrightarrow \quad [\![\tau]\!]$$

Difficulty.

- Equation

$$Y\,t \quad = \quad t\,(Y\,t)$$

imposes

$$[\![Y]\!] \quad : \quad (S \to S) \quad \longrightarrow \quad S$$

A Denotational Semantics for PCF ?

We would like

- each type τ to be interpreted as a set $[\![\tau]\!]$,
- a program t of type say $\sigma \to \tau$ to be interpreted as a function

$$[\![t]\!] \quad : \quad [\![\sigma]\!] \quad \longrightarrow \quad [\![\tau]\!]$$

Difficulty.

- Equation

$$Y\, t \quad = \quad t\,(Y\, t)$$

imposes

$$[\![Y]\!] \quad : \quad (S \to S) \quad \longrightarrow \quad S$$

Traditional Solution.

- Restrict $S \to S$ to the continuous functions for a suitable topology (cpos, Scott domains, etc).

Simpler Settings for Semantics

Gödel's System T.

- Restrict Y to recursion over $\mathbb{N}$:

$$\begin{array}{lcl} \mathtt{rec}\ u\ v\ \underline{0} & = & u \\ \mathtt{rec}\ u\ v\ t\mathtt{+1} & = & v\ t\ (\mathtt{rec}\ u\ v\ t) \end{array}$$

Simpler Settings for Semantics

Gödel's System T.

- Restrict Y to recursion over $\mathbb{N}$:

$$\begin{array}{lcl} \mathbf{rec}\ u\ v\ \underline{0} & = & u \\ \mathbf{rec}\ u\ v\ t\mathbf{+1} & = & v\ t\ (\mathbf{rec}\ u\ v\ t) \end{array}$$

- Allows to see important basic techniques in a simple setting.

Simpler Settings for Semantics

Gödel's System T.

- Restrict Y to recursion over $\mathbb{N}$:

$$\begin{array}{lcl} \mathbf{rec}\ u\ v\ \underline{0} & = & u \\ \mathbf{rec}\ u\ v\ t\mathbf{+1} & = & v\ t\ (\mathbf{rec}\ u\ v\ t) \end{array}$$

- Allows to see important basic techniques in a simple setting.

Simply Typed λ-Calculus.

- System T without **nat**.
- A kernel language without recursion.

Simpler Settings for Semantics

Gödel's System T.

- Restrict Y to recursion over $\mathbb{N}$:

$$\begin{array}{lcl} \mathtt{rec}\ u\ v\ \underline{0} & = & u \\ \mathtt{rec}\ u\ v\ t\mathtt{+1} & = & v\ t\ (\mathtt{rec}\ u\ v\ t) \end{array}$$

- Allows to see important basic techniques in a simple setting.

Simply Typed λ-Calculus.

- System T without **`nat`**.
- A kernel language without recursion.
- Simple instance of the *Curry-Howard-Lambek Correspondence*:

Simpler Settings for Semantics

Gödel's System T.

- Restrict Y to recursion over $\mathbb{N}$:

$$\begin{array}{lcl} \mathtt{rec}\ u\ v\ \underline{0} & = & u \\ \mathtt{rec}\ u\ v\ t\mathtt{+1} & = & v\ t\ (\mathtt{rec}\ u\ v\ t) \end{array}$$

- Allows to see important basic techniques in a simple setting.

Simply Typed λ-Calculus.

- System T without **nat**.
- A kernel language without recursion.
- Simple instance of the *Curry-Howard-Lambek Correspondence*:

		Intuitionistic $(\Rightarrow, \wedge)$-Logic		Cartesian Closed Categories
Types	$\equiv$	Formulae	$\equiv$	Objects
Programs	$\equiv$	Proofs	$\equiv$	Morphisms

Indicative Outline

Indicative Outline

Logical Foundations of Programming Languages (C. RIBA)

Indicative Outline

Logical Foundations of Programming Languages (C. Riba)

Gödel's System T.

- Set-Theoretic Semantics.

Indicative Outline

Logical Foundations of Programming Languages (C. RIBA)

Gödel's System T.

- Set-Theoretic Semantics.

Categorical Semantics of the Simply Typed λ-Calculus.

- Basic Notions of Category Theory.
- Cartesian Closed Categories.
- (Curry-Howard Correspondence.)

Indicative Outline

Logical Foundations of Programming Languages (C. RIBA)

Gödel's System T.

- Set-Theoretic Semantics.

Categorical Semantics of the Simply Typed λ-Calculus.

- Basic Notions of Category Theory.
- Cartesian Closed Categories.
- (Curry-Howard Correspondence.)

PCF.

- CPOs and Scott-Continuity.
- Denotational Semantics.

Indicative Outline

Logical Foundations of Programming Languages (C. RIBA)

Gödel's System T.

- Set-Theoretic Semantics.

Categorical Semantics of the Simply Typed λ-Calculus.

- Basic Notions of Category Theory.
- Cartesian Closed Categories.
- (Curry-Howard Correspondence.)

PCF.

- CPOs and Scott-Continuity.
- Denotational Semantics.

Category Theory.

- Adjunctions.
- Monads.
- ...