Advanced Semantics of Programming Languages

Pierre CLAIRAMBAULT & Colin RIBA

LIP - ENS de Lyon

Course 01
09/11
A Naive Introduction
(based on simple examples)
The First Example
Consider the programs

\[
\text{foo}(f:\text{int}\to\text{int}):
\begin{align*}
\text{return } f(5) + f(5)
\end{align*}
\]

and

\[
\text{bar}(f:\text{int}\to\text{int}):
\begin{align*}
a &= f(5) \\
\text{return } a + a
\end{align*}
\]
The First Example
Consider the programs

```
foo(f:int->int):
    return f(5) + f(5)
```
and

```
bar(f:int->int):
    a = f(5)
    return a + a
```

Are these two programs equivalent?
The First Example
Consider the programs

\[
\text{foo}(f:\text{int}\rightarrow\text{int}): \\
\quad \text{return } f(5) + f(5)
\]

and

\[
\text{bar}(f:\text{int}\rightarrow\text{int}): \\
\quad a = f(5) \\
\quad \text{return } a + a
\]

Are these two programs equivalent?
▶ They are not equivalent if \( f \) can access a global reference.
The First Example
Consider the programs

\[
\begin{align*}
\text{foo}(f: \text{int}\rightarrow\text{int}) : \quad & \text{return } f(5) + f(5) \\
\text{bar}(f: \text{int}\rightarrow\text{int}) : \quad & a = f(5) \\
& \text{return } a + a
\end{align*}
\]

Are these two programs equivalent?
- They are not equivalent if \( f \) can access a global reference.
- They are equivalent if \( f \) behaves as a function, say

\[
[f] : [\text{int}] \rightarrow [\text{int}]
\]

where \([\text{int}]\) is a set representing the type \text{int}.
The First Example
Consider the programs

\[
\begin{aligned}
\text{foo}(f: \text{int} \to \text{int}): \\
&\quad \text{return } f(5) + f(5)
\end{aligned}
\quad \text{and} \quad
\begin{aligned}
\text{bar}(f: \text{int} \to \text{int}): \\
&\quad a = f(5) \\
&\quad \text{return } a + a
\end{aligned}
\]

Are these two programs equivalent?
- They are not equivalent if \( f \) can access a global reference.
- They are equivalent if \( f \) behaves as a function, say

\[
[f] : [\text{int}] \rightarrow [\text{int}]
\]

where \([\text{int}]\) is a set representing the type \(\text{int}\).

Objectives of the course.
- Mathematical models of programming languages
  (denotational semantics, category theory, type systems).
The First Example

Consider the programs

\[
\text{foo}(f:\text{int}\rightarrow\text{int}) : \quad \text{return } f(5) + f(5)
\]

and

\[
\text{bar}(f:\text{int}\rightarrow\text{int}) : \quad a = f(5) \quad \text{return } a + a
\]

Are these two programs equivalent?

▶ They are not equivalent if \( f \) can access a global reference.
▶ They are equivalent if \( f \) behaves as a \textbf{function}, say

\[
[f] : [\text{int}] \rightarrow [\text{int}]
\]

where \([\text{int}]\) is a set representing the type \text{int}.

Objectives of the course.

▶ Mathematical models of programming languages
  (denotational semantics, category theory, type systems).

Methodology of the course.

▶ Begin with simple approaches.
▶ Then progressively model more complex behaviours.
The First Example
Consider the programs

\[
\begin{align*}
\text{foo}(f: \text{int} \rightarrow \text{int}) : \\
& \quad \text{return } f(5) + f(5)
\end{align*}
\]

and

\[
\begin{align*}
\text{bar}(f: \text{int} \rightarrow \text{int}) : \\
& \quad a = f(5) \\
& \quad \text{return } a + a
\end{align*}
\]

Are these two programs equivalent?

- They are not equivalent if \( f \) can access a global reference.
- They are equivalent if \( f \) behaves as a function, say

\[
[f] : [\text{int}] \rightarrow [\text{int}]
\]

where \([\text{int}]\) is a set representing the type \text{int}.

Objectives of the course.
- Mathematical models of programming languages (denotational semantics, category theory, type systems).

Methodology of the course.
- Begin with simple approaches.
- Then progressively model more complex behaviours.

Now:
- A naive introduction to some basic ideas.
Types as Sets

Assume types, say \texttt{int}, \texttt{bool}, are to be interpreted as sets $\mathbb{J}_{\texttt{int}}$, $\mathbb{J}_{\texttt{bool}}$. 

Question.

Can we assume $\mathbb{J}_{\texttt{bool}} := \{ \text{true}, \text{false} \}$ (1)

Answer.

Consider the non-terminating program

\begin{verbatim}
loop (b:bool):
  while true:
    skip
  return true
\end{verbatim}

If $\mathbb{J}_{\texttt{bool}}$ is as in (1), then we can not have $\mathbb{J}_{\texttt{loop}} := \mathbb{J}_{\texttt{bool}}$.

We shall therefore represent divergence and assume $\mathbb{J}_{\texttt{bool}} := \{ \bot, \text{true}, \text{false} \}$ (\(\bot\) = divergence)

We can then have, as expected:

$\mathbb{J}_{\texttt{loop}}(a) = \bot$ (for all $a \in \mathbb{J}_{\texttt{bool}}$)
Types as Sets

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

Question.

Can we assume  

$$[\texttt{bool}] := \{\texttt{true}, \texttt{false}\} \quad (1)$$
Types as Sets

Assume types, say \( \texttt{int}, \texttt{bool} \), are to be interpreted as sets \([\texttt{int}], [\texttt{bool}]\).

**Question.**
Can we assume 
\[
[\texttt{bool}] := \{\texttt{true, false}\}
\]  
(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 \([\text{int}], [\text{bool}]\).

Question.

Can we assume

\[
[\text{bool}] \ := \ \{\text{true}, \text{false}\}
\]  

(1)

Answer.

Consider the non-terminating program

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

If \([\text{bool}]\) is as in (1), then we can not have

\[
[\text{loop}] : [\text{bool}] \rightarrow [\text{bool}]
\]
Types as Sets

Assume types, say \texttt{int}, \texttt{bool}, are to be interpreted as sets \([\texttt{int}], [\texttt{bool}]\).

Question.

Can we assume

\[
[\texttt{bool}] := \{\texttt{true}, \texttt{false}\}
\]  \hfill (1)

Answer.

Consider the non-terminating program

\begin{verbatim}
loop (b:bool):
    while true:
        skip
    return true
\end{verbatim}

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

\[
[\text{loop}] : [\texttt{bool}] \rightarrow [\texttt{bool}]
\]

We shall therefore represent divergence and assume

\[
[\texttt{bool}] := \{\bot, \texttt{true}, \texttt{false}\}
\]

\((\bot \text{ “=}” \text{ divergence})\)

We can then have, as expected:

\[
[\text{loop}](a) = \bot \quad \text{(for all } a \in [\texttt{bool}])
\]
A Taste of Finitary PCF

Motivation.

▶ A simple language to discuss

\[
[\text{bool}] := \{\bot, \text{true}, \text{false}\}
\]
A Taste of Finitary PCF

Motivation.

A simple language to discuss

$$\llbracket \text{bool} \rrbracket := \{\bot, \text{true}, \text{false}\}$$

The Language of Finitary PCF.

$$\begin{align*}
\tau, \sigma & \ ::= \ \text{bool} \mid \sigma \rightarrow \tau \\
t, u & \ ::= \ t u \mid \text{fun} \ x \rightarrow t \mid \text{true} \mid \text{false} \mid \text{if } t \text{ then } u \text{ else } v \mid \Omega
\end{align*}$$
A Taste of Finitary PCF

Motivation.

- A simple language to discuss

\[
\lbrack \text{bool} \rbrack := \{ \perp, \text{true}, \text{false} \}
\]

The Language of Finitary PCF.

\[
\begin{align*}
\tau, \sigma & := \text{bool} \mid \sigma \rightarrow \tau \\
t, u & := t \, u \mid \text{fun } x \rightarrow t \mid \text{true} \mid \text{false} \mid \text{if } t \text{ then } u \text{ else } v \mid \Omega
\end{align*}
\]

- Purely functional language with Booleans and divergence (\(\Omega\)).
A Taste of Finitary PCF

Motivation.

- A simple language to discuss

\[ [\text{bool}] := \{\bot, \text{true}, \text{false}\} \]

The Language of Finitary PCF.

\[ \tau, \sigma ::= \text{bool} \mid \sigma \rightarrow \tau \]

\[ t, u ::= t \ u \mid \text{fun} \ x \rightarrow t \mid \text{true} \mid \text{false} \mid \text{if} \ t \ \text{then} \ u \ \text{else} \ v \mid \Omega \]

- Purely functional language with Booleans and divergence (\(\Omega\)).

We assume call-by-name evaluation:

\[
(\text{fun} \ x \rightarrow t)u = t[u/x] \\
\text{if} \ \text{true} \ \text{then} \ t \ \text{else} \ u = t \\
\text{if} \ \text{false} \ \text{then} \ t \ \text{else} \ u = u
\]
Example.
Consider the two following or programs:

\[
\text{or}_l := \text{fun } a, b \rightarrow \\
\quad \text{if } a \text{ then } a \text{ else } b
\]

\[
\text{vs}
\]

\[
\text{or}_r := \text{fun } a, b \rightarrow \\
\quad \text{if } b \text{ then } b \text{ else } a
\]

Questions.
▶ What are the functions \( J_{\text{or}_l} K \) and \( J_{\text{or}_r} K \)?
▶ Are the programs \( \text{or}_l \) and \( \text{or}_r \) equivalent?

Example.
Consider, for \( b \in \{\text{true}, \text{false}\} \), the program

\[
taste_b := \text{fun } f \rightarrow \\
\quad \text{if } f(\text{true}, \Omega) \text{ and } f(\Omega, \text{true}) \text{ and } \neg(f(\text{false}, \text{false})) \text{ then } b \text{ else } \text{true}
\]

Questions.
▶ Do we have \( J_{\text{taste}_\text{true}} K = J_{\text{taste}_\text{false}} K \)?
▶ Are \( \text{taste}_\text{true} \) and \( \text{taste}_\text{false} \) equivalent?
Example.
Consider the two following or programs:

\[
\begin{align*}
or_l & := \text{fun } a, b \rightarrow \text{if } a \text{ then } a \text{ else } b \\
\end{align*}
\]

\[
\begin{align*}
or_r & := \text{fun } a, b \rightarrow \text{if } b \text{ then } b \text{ else } a \\
\end{align*}
\]

Questions.

- What are the functions \( [or_1] \), \( [or_r] \) ?
- Are the programs \( or_1 \) and \( or_r \) equivalent?
Example.
Consider the two following `or` programs:

\[
\begin{align*}
\text{or}_l & := \text{fun } a, b \rightarrow \\
& \quad \text{if } a \text{ then } a \text{ else } b \\
\text{or}_r & := \text{fun } a, b \rightarrow \\
& \quad \text{if } b \text{ then } b \text{ else } a
\end{align*}
\]

Questions.
- What are the functions \([\text{or}_l]\), \([\text{or}_r]\) ?
- Are the programs \(\text{or}_l\) and \(\text{or}_r\) equivalent ?

Example.
Consider, for \(b \in \{\text{true}, \text{false}\}\), the program

\[
\begin{align*}
taste_b & := \text{fun } f \rightarrow \\
& \quad \text{if } f(\text{true}, \Omega) \text{ and } \\
& \quad f(\Omega, \text{true}) \text{ and } \\
& \quad \text{not}(f(\text{false}, \text{false})) \text{ then } b \\
& \quad \text{else } \text{true}
\end{align*}
\]
Example.
Consider the two following or programs:

\[
\begin{align*}
\text{or}_l & := \text{fun } a, b \rightarrow \\
& \quad \text{if } a \text{ then } a \text{ else } b
\end{align*}
\]

\[
\begin{align*}
\text{or}_r & := \text{fun } a, b \rightarrow \\
& \quad \text{if } b \text{ then } b \text{ else } a
\end{align*}
\]

Questions.

- What are the functions \([\text{or}_l]\), \([\text{or}_r]\) ?
- Are the programs \text{or}_l and \text{or}_r equivalent ?

Example.
Consider, for \(b \in \{\text{true}, \text{false}\}\), the program

\[
\begin{align*}
\text{taste}_b & := \text{fun } f \rightarrow \\
& \quad \text{if } f(\text{true}, \Omega) \text{ and } \\
& \quad f(\Omega, \text{true}) \text{ and } \\
& \quad \text{not}(f(\text{false}, \text{false})) \text{ then } b \\
& \quad \text{else } \text{true}
\end{align*}
\]

Questions.

- Do we have \([\text{taste_true}] = [\text{taste_false}]\) ?
- Are \text{taste_true} and \text{taste_false} equivalent ?
A Taste of PCF

Motivation.

- Extend Finitary PCF with general recursion.
- Mathematically cleaner if an infinite type is assumed (say the natural numbers).
A Taste of PCF

Motivation.
▶ Extend Finitary PCF with general recursion.
▶ Mathematically cleaner if an infinite type is assumed (say the natural numbers).

The Language of PCF.

\[
\begin{align*}
\tau, \sigma & ::= \ldots \mid \text{nat} \\
t, u & ::= \ldots \mid t+1 \mid t-1 \mid z? \mid Y \mid n \quad \text{(for each } n \in \mathbb{N})
\end{align*}
\]

▶ \(Y\) is a \textit{fixpoint} combinator:

\[
Y \; t \; = \; t \; (\; Y \; t \;)
\]
A Taste of PCF

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

The Language of PCF.

\[ \tau, \sigma ::= \ldots | \text{nat} \]

\[ t, u ::= \ldots | t+1 | t-1 | z? | Y | n \quad \text{(for each } n \in \mathbb{N}) \]

- Y is a fixpoint combinator:
  \[ Y \, t = t \,(Y \, t) \]

Examples.
- We could have defined
  \[ \Omega ::= Y(\text{fun } x \rightarrow x) \]
A Taste of PCF

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

The Language of PCF.

\[
\begin{align*}
\tau, \sigma & ::= \ldots \mid \text{nat} \\
t, u & ::= \ldots \mid t+1 \mid t-1 \mid z? \mid Y \mid n \quad \text{(for each } n \in \mathbb{N})
\end{align*}
\]

- \(Y\) is a \textit{fixpoint} combinator:
  \[
  Y \ t = t \ (Y \ t)
  \]

Examples.
- We could have defined
  \[
  \Omega := Y (\text{fun } x \to x)
  \]
- Addition
  \[
  \begin{align*}
  \text{add } 0 \ u & = u \\
  \text{add } t+1 \ u & = (\text{add } t \ u) + 1
  \end{align*}
  \]
A Taste of PCF

Motivation.
▶ Extend Finitary PCF with general recursion.
▶ Mathematically cleaner if an infinite type is assumed (say the natural numbers).

The Language of PCF.
\[
\begin{align*}
\tau, \sigma & ::= \ldots | \text{nat} \\
t, u & ::= \ldots | t+1 | t-1 | z? | Y | n \quad (\text{for each } n \in \mathbb{N})
\end{align*}
\]
▶ \(Y\) is a fixpoint combinator:
\[
Y t = t (Y t)
\]

Examples.
▶ We could have defined
\[
\Omega := Y (\text{fun } x \rightarrow x)
\]
▶ Addition
\[
\begin{align*}
\text{add } 0 u & = u \\
\text{add } t+1 u & = (\text{add } t u)+1
\end{align*}
\]
can be defined as
\[
\text{add} := Y \text{add}_\text{rec}
\]
where
\[
\begin{align*}
\text{add}_\text{rec} & := \text{fun } f, x, y \rightarrow \\
& \quad \text{if } (z? x) \text{ then } y \text{ else } (f x-1 y)+1
\end{align*}
\]
A Denotational Semantics for PCF?
A Denotational Semantics for PCF?

We Would Like

- each type $\tau$ to be interpreted as a set $[\tau]$,
A Denotational Semantics for PCF?

We Would Like

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

$$[t] : [\sigma] \rightarrow [\tau]$$
A Denotational Semantics for PCF?

We Would Like

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

\[
[t] : [\sigma] \rightarrow [\tau]
\]

Difficulty.

- Equation

\[
Yt = t(Yt)
\]

imposes

\[
[Y] : (S \rightarrow S) \rightarrow S
\]
A Denotational Semantics for PCF ?

We Would Like

▶ each type \( \tau \) to be interpreted as a set \([\tau]\),
▶ a program \( t \) of type say \( \sigma \to \tau \) to be interpreted as a function

\[
[t] : [\sigma] \rightarrow [\tau]
\]

Difficulty.

▶ Equation

\[
Y \ t = t (Y \ t)
\]

imposes

\[
[Y] : (S \to S) \rightarrow S
\]

Traditional Solution.

▶ Restrict \( S \to S \) to the continuous functions for a suitable topology (cpos, Scott domains, etc).
A Denotational Semantics for PCF?

We Would Like

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

$$[t] : [\sigma] \rightarrow [\tau]$$

Difficulty.

- Equation

$$Y \ t = t (Y \ t)$$

imposes

$$[Y] : (S \rightarrow S) \rightarrow S$$

Traditional Solution.

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

Gödel’s System T.

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

$$\text{rec } u \ v \ 0 = u$$
$$\text{rec } u \ v \ t + 1 = v \ t (\text{rec } u \ v \ t)$$
A Denotational Semantics for PCF?

We Would Like

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

$$[t] : [\sigma] \rightarrow [\tau]$$

Difficulty.

- Equation

$$Y\ t = t\ (Y\ t)$$

imposes

$$[Y] : (S \rightarrow S) \rightarrow S$$

Traditional Solution.

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

Gödel’s System T.

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

$$\text{rec } u \nu 0 = u$$
$$\text{rec } u \nu t + 1 = \nu t (\text{rec } u \nu t)$$

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

Courses 1–6:

- Set-theoretic semantics of System T.
- Denotational semantics of PCF (cpos, logical relations).
- Further topics among:
  - Polymorphism (Girard-Reynolds System F).
  - Recursive types.
  - Intersection types.
  - Scott domains and PCF definability.
  - Results on the set-theoretic semantics of the simply-typed $\lambda$-calculus.
Indicative Outline

Courses 1–6: (C. Riba)
- Set-theoretic semantics of System T.
- Denotational semantics of PCF (cpos, logical relations).
- Further topics among:
  - Polymorphism (Girard-Reynolds System F).
  - Recursive types.
  - Intersection types.
  - Scott domains and PCF definability.
  - Results on the set-theoretic semantics of the simply-typed \( \lambda \)-calculus.

Courses 7–12: (P. Clairambault)
- Categories, functors and natural transformations.
- Cartesian closed categories and the \( \lambda \)-calculus.
- Monads.
- Further topics among:
  - Categorical models of linear logic.
  - Game semantics.
Indicative Outline

Courses 1–6: (C. Riba)

- Set-theoretic semantics of System T.
- Denotational semantics of PCF (cpos, logical relations).
- Further topics among:
  - Polymorphism (Girard-Reynolds System F).
  - Recursive types.
  - Intersection types.
  - Scott domains and PCF definability.
  - Results on the set-theoretic semantics of the simply-typed $\lambda$-calculus.

Courses 7–12: (P. Clairambault)

- Categories, functors and natural transformations.
- Cartesian closed categories and the $\lambda$-calculus.
- Monads.
- Further topics among:
  - Categorical models of linear logic.
  - Game semantics.

Courses 13–: Survey of Some Active Research Topics.