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

\begin{verbatim}
foo(f:int->int):
  return f(5) + f(5)
\end{verbatim}

and

\begin{verbatim}
bar(f:int->int):
  a = f(5)
  return a + a
\end{verbatim}

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 \( JfK \):

\( J \) \( \text{int} \) \( K \) \( \rightarrow \) \( J \) \( \text{int} \) \( K \)

where \( J \) \( \text{int} \) \( K \) 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

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

and

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

Are these two programs equivalent?
The First Example

Consider the programs

<table>
<thead>
<tr>
<th>foo (f : int → int):</th>
<th>bar (f : int → int):</th>
</tr>
</thead>
<tbody>
<tr>
<td>return f(5) + f(5)</td>
<td>a = f(5)</td>
</tr>
<tr>
<td></td>
<td>return a + a</td>
</tr>
</tbody>
</table>

Are these two programs equivalent?
- They are not equivalent if $f$ can access a global reference.
The First Example

Consider the programs

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

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 \textbf{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

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

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

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

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

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

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.
The First Example

Consider the programs

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

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 progressivly 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 \([\texttt{int}]\), \([\texttt{bool}]\).
Types as Sets

- Assume types, say `int`, `bool`, are to be interpreted as sets `[int]`, `[bool]`.

**Question.**
- Can we assume
  \[
  [\text{bool}] := \{ \text{true}, \text{false} \} 
  \] (1)
Types as Sets

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

**Question.**
- Can we assume

\[
[\text{bool}] := \{\text{true, false}\} \tag{1}
\]

**Answer.**
- Consider the non-terminating program

```python
loop (b:bool):
    while true:
        skip
    return true
```
Types as Sets

- Assume types, say `int`, `bool`, are to be interpreted as sets `[int]`, `[bool]`.

**Question.**
- Can we assume

  \[ [\text{bool}] \ := \ \{\text{true, false}\} \quad (1) \]

**Answer.**
- Consider the non-terminating program

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

- If `[bool]` is as in (1), then we can not have

  \[ [\text{loop}] : [\text{bool}] \rightarrow [\text{bool}] \]
Naive Introduction

Types as Sets

Assume types, say `int`, `bool`, are to be interpreted as sets `[int]`, `[bool]`.

Question.

Can we assume

\[
{\text{[bool] := \{true, false\}}} \quad \text{(1)}
\]

Answer.

Consider the non-terminating program

\[
\text{loop (b:bool):}
\]
\[
\text{while true:}
\]
\[
\text{skip}
\]
\[
\text{return true}
\]

If `[bool]` is as in (1), then we can not have

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

We shall therefore represent divergence and assume

\[
{\text{[bool] := \{\bot, true, false\} \quad (\bot \text{ "=" divergence)}}
\]

We can then have, as expected:

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

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

The Language of Finitary PCF.

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

Motivation.

▶ A simple language to discuss

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

The Language of Finitary PCF.

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

▶ Purely functional language with Booleans and divergence (Ω).
A Taste of Finitary PCF

Motivation.

▶ A simple language to discuss

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

The Language of Finitary PCF.

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

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

Example.

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

vs

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

Motivation.

A simple language to discuss

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

The Language of Finitary PCF.

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

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

Example.

\[
\begin{align*}
or_1 & := \text{fun} \ a, b \rightarrow \text{if} \ a \ \text{then} \ a \ \text{else} \ b \\
or_r & := \text{fun} \ a, b \rightarrow \text{if} \ b \ \text{then} \ b \ \text{else} \ a
\end{align*}
\]

Questions.

What are the functions \([or_1], [or_r]\) ?

Are the programs \(or_1\) and \(or_r\) equivalent ?
A Theoretical Example

Assume *call-by-name* evaluation:

\[
(f \text{un } x \rightarrow t)u = t[u/x]
\]

\[
\text{if true then } t \text{ else } u = t
\]

\[
\text{if false then } t \text{ else } u = u
\]
A Theoretical Example

Assume *call-by-name* evaluation:

\[
(fun \ x \ \rightarrow \ t)u = t[u/x]
\]

\[
\text{if true then } t \ \text{else } u = t
\]

\[
\text{if false then } t \ \text{else } u = u
\]

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

\[
taste_b := \text{fun } f \rightarrow
\]

\[
\text{if } f(\text{true}, \ \Omega) \ \text{and}
\]

\[
f(\Omega, \ \text{true}) \ \text{and}
\]

\[
\text{not}(f(\text{false}, \ \text{false}))
\]

\[
\text{then } \ b
\]

\[
\text{else } \text{true}
\]

Questions.

▶ Do we have \(J_{taste_{true}} = J_{taste_{false}}\)?

▶ Are \(taste_{true}\) and \(taste_{false}\) equivalent?
A Theoretical Example

Assume *call-by-name* evaluation:

\[
(f\,x \to t)\,u = t[u/x]
\]

\[
\text{if true then } t \text{ else } u = t
\]

\[
\text{if false then } t \text{ else } u = u
\]

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

\[
taste_b := \text{fun } f ->
\begin{align*}
\text{if } f(\text{true, } \Omega) \text{ and } \\
\text{f}(\Omega, \text{true}) \text{ and } \\
\text{not}(f(\text{false, } \text{false}))
\end{align*}
\]

\[
\text{then } b \\
\text{else true}
\]

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.
- 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.

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

- \(Y\) is a 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.

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

- \( Y \) 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.
- $\tau, \sigma ::= \ldots \mid \text{nat}$
- $t, u ::= \ldots \mid t+1 \mid t-1 \mid z? \mid Y \mid n$ (for each $n \in \mathbb{N}$)
- $Y$ is a fixpoint combinator:
  $$Y t = t(Y t)$$

Examples.
- We could have defined
  $$\Omega := Y(\text{fun } x \rightarrow x)$$
- 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.

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

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

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

We would like each type \( \tau \) to be interpreted as a set \( J_\tau \), each program \( t \) of type \( \sigma \rightarrow \tau \) to be interpreted as a function \( J_t : J_\sigma \rightarrow J_\tau \).

Difficulty.

Equation \( Y_t = t(Y_t) \) imposes \( J_Y : (S \rightarrow S) \rightarrow S \).

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 \( \mathbb{N} \):

\[
\text{rec}_{u} v 0 = u \\
\text{rec}_{u} v t + 1 = v t(\text{rec}_{u} v t)
\]

Allows to see important techniques in a simple setting.
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 $\sigma \to \tau$ 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 $\llbracket \tau \rrbracket$,
- a program $t$ of type say $\sigma \rightarrow \tau$ to be interpreted as a function

$$\llbracket t \rrbracket : \llbracket \sigma \rrbracket \rightarrow \llbracket \tau \rrbracket$$

Difficulty.

- Equation

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

imposes

$$\llbracket Y \rrbracket : (S \rightarrow S) \rightarrow S$$
A Denotational Semantics for PCF?

We Would Like

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

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

Difficulty.

- Equation

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

imposes

$$\lfloor Y \rfloor : (S \rightarrow S) \rightarrow S$$

Traditional Solution.

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

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

$$\llbracket t \rrbracket : \llbracket \sigma \rrbracket \rightarrow \llbracket \tau \rrbracket$$

Difficulty.

- Equation

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

imposes

$$\llbracket Y \rrbracket : (S \rightarrow S) \rightarrow S$$

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 $\llbracket \tau \rrbracket$,
- a program $t$ of type say $\sigma \to \tau$ to be interpreted as a function

$$\llbracket t \rrbracket : \llbracket \sigma \rrbracket \to \llbracket \tau \rrbracket$$

Difficulty.

- Equation

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

imposes

$$\llbracket Y \rrbracket : (S \to S) \to S$$

Traditional Solution.

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

Gödel’s System T.

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

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

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

Courses 1–3: Basic Notions.

- Set-theoretic semantics of System $T$.
- Denotational semantics of PCF (cpos, logical relations).
- Further topics (if time permits):
  - Scott domains and PCF definability.
  - Results on the set-theoretic semantics of the simply-typed $\lambda$-calculus.
Rough Outline

Courses 1–3: Basic Notions. (C. Riba)
- Set-theoretic semantics of System $T$.
- Denotational semantics of PCF (cpos, logical relations).
- Further topics (if time permits):
  - Scott domains and PCF definability.
  - Results on the set-theoretic semantics of the simply-typed $\lambda$-calculus.

Courses 4–6: Introduction to Category Theory. (P. Clairambault)
- Categories, functors and natural transformations.
- Cartesian closed categories and the $\lambda$-calculus.
Rough Outline

Courses 1–3: Basic Notions. (C. Riba)

- Set-theoretic semantics of System $T$.
- Denotational semantics of PCF (cposs, logical relations).
- Further topics (if time permits):
  - Scott domains and PCF definability.
  - Results on the set-theoretic semantics of the simply-typed $\lambda$-calculus.

Courses 4–6: Introduction to Category Theory. (P. Clairambault)

- Categories, functors and natural transformations.
- Cartesian closed categories and the $\lambda$-calculus.

Courses 7–9: Selected Topics.

- Monads.
- Polymorphism.
- Recursive Types.
Rough Outline

Courses 1–3: Basic Notions. (C. Riba)

▶ Set-theoretic semantics of System $T$.
▶ Denotational semantics of PCF (cpos, logical relations).
▶ Further topics (if time permits):
  ▶ Scott domains and PCF definability.
  ▶ Results on the set-theoretic semantics of the simply-typed $\lambda$-calculus.

Courses 4–6: Introduction to Category Theory. (P. Clairambault)

▶ Categories, functors and natural transformations.
▶ Cartesian closed categories and the $\lambda$-calculus.

Courses 7–9: Selected Topics.

▶ Monads.
▶ Polymorphism.
▶ Recursive Types.

Courses 10–: Survey of Some Active Research Topics.