Typage

lecture 1

2018-2019
Giovanni Bernardi, gioXYZirif.fr
http://www.irif.fr/~gio/index.xhtml
Université Paris Diderot
Practical information

Teaching

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lectures</td>
<td>wednesdays</td>
<td>2036</td>
<td>09h30 – 11h00</td>
<td></td>
</tr>
<tr>
<td>TP</td>
<td>wednesdays</td>
<td>2038</td>
<td>11h00 – 12h30</td>
<td></td>
</tr>
</tbody>
</table>

No lecture/TP 16 Jan!!!

Final grade

▶ Exam 40%
▶ Projet 60%

1. One .ml file that contains ocaml
2. Type inference + unification algorithms finite types
3. Type unification algorithm recursive types
http://www.irif.fr/~gio/index.xhtml

“Types and Programming Languages”  “Introduction To Bisimulation and Coinduction”

B. Pierce  D. Sangiori

few chapters  one chapter

don’t be shy :-)  Do ask questions!!!

feedback helps
Overall aim
Make you understand (co)induction.
Show you different
▶ standpoints on the subject matter
▶ applications

First part of the lectures
1. Mini functional language
2. Operational semantics
3. Monomorphic types
4. Polymorphic types
Who introduced $\lambda$-calculus?

An Unsolvable Problem of Elementary Number Theory
Alonzo Church, 1936

annus mirabilis CS

1936

https://functionaljobs.com/
A functional language

Syntax

\[
M, N ::= \ x \quad \text{(variable)} \\
\ ct \quad \text{(constant)} \\
\langle M, N\rangle \quad \text{(pair)} \\
M \ N \quad \text{(application)} \\
\lambda x.M \quad \text{(abstraction)} \\
\text{let } x = M \text{ in } N \quad \text{(let)}
\]

Some constants:
\(\text{fst, snd, fix, ifthenelse, +, *, \ldots, true, false, 0, 1, 2, 3, \ldots}\)

- Do you understand the syntax above?
- Note circularity: \(M, N\) appear left and right of \(::=\)

What does a program do? And why?
Notations

\[ M_1 M_2 \ldots M_n \equiv (\ldots((M_1 M_2)M_3)\ldots M_{n-1}) M_n \]
\[ N \tilde{M} \equiv (\ldots(((N M_1) M_2) M_3) \ldots M_{n-1}) M_n \]
\[ M + N \equiv +\langle M, N \rangle \]
\[ \text{if } E \text{ then } M \text{ else } N \equiv \text{ifthenelse}\langle E, \langle M, N \rangle \rangle \]
Free variables

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(ct) &= \emptyset \\
FV(\langle M, N \rangle) &= FV(M) \cup FV(N) \\
FV(M \ N) &= FV(M) \cup FV(N) \\
FV(\lambda x. M) &= FV(M) \setminus \{x\} \\
FV(\text{let } x = M \text{ in } N) &= FV(M) \cup FV(N) \setminus \{x\}
\end{align*}
\]

A variable \( x \) is free in \( M \) if \( x \in FV(M) \).
A term \( M \) is closed iff it has no free variable, i.e. \( FV(M) = \emptyset \).

Example

\begin{itemize}
\item \( M = \lambda z.((\lambda x.x \ z)(\lambda y.y)) \) is closed \( FV(M) = \emptyset \)
\item \( M = (\lambda x.x \ z)(\lambda y.y) \) is not closed \( FV(M) = \{z\} \)
\end{itemize}

Note **circularity**: \( FV \) appears left and right of \( = \)
Bound variables

\[
BV(x) = \emptyset \\
BV(ct) = \emptyset \\
BV(\langle M, N \rangle) = BV(M) \cup BV(N) \\
BV(M \ N) = BV(M) \cup BV(N) \\
BV(\lambda x. M) = BV(M) \cup \{x\} \\
BV(\text{let } x = M \text{ in } N) = BV(M) \cup BV(N) \cup \{x\}
\]

A variable \( x \) is **bound** in \( M \) if \( x \in BV(M) \).

**Example**

- \( x \) not bound in \( x \)
- \( x \) bound in \( \lambda x. x \)
- \( x \) free and bound in \( x (\lambda x. x) \)

Note **circularity**: \( BV \) appears left and right of \( = \)
Alpha-conversion

\[=_{\alpha}\] equivalence allowing renaming bound variables.

**Example**

\[\begin{align*}
&\rightarrow x (\lambda x.x y) =_{\alpha} x (\lambda z.z y) \\
&\rightarrow \text{let } x = x' \text{ in } x y =_{\alpha} \text{let } z = x' \text{ in } z y \\
&\rightarrow \text{more in general } \int x^2 \, dx =_{\alpha} \int y^2 \, dy
\end{align*}\]

**Theorem**

*For every term* \(M\) *there is a term* \(M'\) *such that*

1. \(M =_{\alpha} M'\)
2. **Barendregt’s Convention:**
   - \(FV(M') \cap BV(M') = \emptyset\).
   - *All the bound variables of* \(M'\) *are distinct.*
Substitution

The application of a substitution $\sigma = \{x_1/M_1, \ldots, x_n/M_n\}$ to a term $M$ is defined by induction\(^1\) as follows:

$$
\begin{align*}
\sigma x_i & = M_i & \text{if } i \in \{1, \ldots, n\} \\
\sigma y & = y & \text{if } y \notin \{x_1, \ldots, x_n\} \\
\sigma ct & = ct \\
\sigma \langle M, N \rangle & = \langle \sigma M, \sigma N \rangle \\
\sigma (M \ N) & = \sigma M \ \sigma N \\
\sigma (\lambda x. M) & = \lambda x. (\sigma M) & \text{if no capture of variables} \\
\sigma (\let x = M \ in \ N) & = \let x = \sigma M \ in \ \sigma N & \text{if no capture of variables}
\end{align*}
$$

Example

$$
\{y/x\}(\lambda x. y \ x) = \lambda z. (\{y/x\}x \ z) = \lambda z. x \ z
$$

\(^1\)Note circularity: $\sigma$ appears left and right of $=$
Operational Semantics
Defining an Operational Semantics

- **Granularity**
  - Big-step
  - Small-step

- **Order of evaluation**
  - Call-by-value
  - Call-by-name
  - ...

- **Inference rules + derivation trees**

  \[
  \begin{array}{c}
  \text{premise}_1 \ldots \text{premise}_n \\
  \hline
  \text{conclusion} \\
  \end{array}
  \]

  Inference rule

  side condition
Big-step Semantics

Each rule\(^2\) completely evaluates the expression to a value.

Sketch

- a arithmetic expression, \(\sigma\) state, \(n\) value
- in state \(\sigma\) the expression \(a\) evaluates to \(n\)\n
\[
(a, \sigma) \Downarrow n
\]

\[
(n, \sigma) \Downarrow n \quad (\sigma(X), \sigma) \Downarrow \sigma(X)
\]

\[
(a_1, \sigma) \Downarrow n_1 \quad (a_2, \sigma) \Downarrow n_2 \quad \frac{\frac{\Downarrow}{(a_1 + a_2, \sigma) \Downarrow n}}{n \text{ is } “n_1 \text{ plus } n_2”}
\]

We write \((a, \sigma) \Downarrow n\) if there exists finite derivation tree \((a, \sigma) \Downarrow n\)

\(^2\)Note circularity: \(\Downarrow\) appears in premises and conclusions of rules
Properties

- Abstract
- Allows to avoid details
- No specification of evaluation order (e.g. \((1 + 3) + (5 - 3)\))
- No specification of control of errors
- No specification of interleaving
It is the purpose of these notes to develop a simple and direct method for specifying the semantics of programming languages. Very little is required in the way of mathematical background; all that will be involved is “symbol-pushing” [...]

Who introduced small-step semantics?

A Structural Approach to Operational Semantics

Gordon Plotkin, 1981
Small-step Semantics

Evaluation is sequence of *state changes* of an abstract machine which terminates when the state cannot be reduced further.

**Sketch**

\[
(X, \sigma) \leadsto (\sigma(X), \sigma) \quad (n_1 + n_2, \sigma) \leadsto (n, \sigma) \quad n \text{ is "} n_1 \text{ plus } n_2 \text{"}
\]

\[
(a_1, \sigma) \leadsto (a'_1, \sigma') \quad (a_2, \sigma) \leadsto (a'_2, \sigma') \\
(a_1 + a_2, \sigma) \leadsto (a'_1 + a_2, \sigma') \quad (n_1 + a_2, \sigma) \leadsto (n_1 + a'_2, \sigma')
\]

We write \((a, \sigma) \leadsto (a', \sigma')\) if there exists **finite** derivation tree \((a, \sigma) \leadsto (a', \sigma')\)

---

\(^3\)Note circularity: \(\leadsto\) appears in premises and conclusions of rules
Properties

- Less abstract
- Specification of order of evaluation

Control of errors: \( \frac{n_2 \neq 0}{n_1 / n_2 \leadsto n} \), where \( n \) is "\( n_1 \) divided by \( n_2 \)".

Interleaving:
\[
\begin{align*}
\langle c_1, \sigma \rangle \leadsto \langle c'_1, \sigma' \rangle \\
\langle c_1 \| c_2, \sigma \rangle \leadsto \langle c'_1 \| c_2, \sigma' \rangle
\end{align*}
\]
The multi-step semantics is given by the relation $t \leadsto^* t'$ which is the reflexive and transitive closure of $t \leadsto t'$.

(P1) $t \leadsto^* t$ for every $t$

(P2) $t \leadsto t'$ implies $t \leadsto^* t'$

(P3) $t \leadsto^* t'$ and $t' \leadsto^* t''$ implies $t \leadsto^* t''$
Properties of the small and big step semantics

- The relation $\rightsquigarrow$ is deterministic.
- The relation $\downarrow$ is deterministic.
- $t \downarrow v$ iff $t \rightsquigarrow^* v$, where $v$ is a ”value”.
Normal Forms

- A **normal form** is a term that cannot be evaluated any further: it is a state where the abstract machine is halted (result of the evaluation).
Big-step versus small-step semantics

- In small-step semantics evaluation stops at errors. In big-step semantics errors occur deeply inside derivation trees.
- The order of evaluation is *explicit* in small-step semantics but *implicit* in big-step semantics.
- Big-step semantics is more abstract, but less precise.
- Small-step semantics allows to make difference between non-termination and "getting stuck".
Reduction Rules

\[(\lambda x. M) \; N \rightarrow M \{ x / N \} \quad \text{\(\beta\)-reduction}\]

\[\text{let } x = N \text{ in } M \rightarrow M \{ x / N \}\]

\[\text{fix } M \rightarrow M \ (\text{fix } M)\]

\[\text{fst} \langle M, N \rangle \rightarrow M\]

\[\text{snd} \langle M, N \rangle \rightarrow N\]

\[\text{if } \text{true} \text{ then } M \text{ else } N \rightarrow M\]

\[\text{if } \text{false} \text{ then } M \text{ else } N \rightarrow N\]

\[\text{if } 0 \text{ then } M \text{ else } N \rightarrow M\]

\[\text{if } n \text{ then } M \text{ else } N \rightarrow N, \ n \neq 0\]

**WARNING!:** The reduction relation \( \rightarrow \) is non-deterministic.