Non-Idempotent Typing Operators, beyond the λ-Calculus
Soutenance de thèse

Pierre VIAL
IRIF (Univ. Paris Diderot and CNRS)

December 7, 2017
x = 1
while (x > 0):
    x = x + 1
transfer(1 000 000 000 $, calyon, my-account)
print("I’m rich now")
x = 1
while (x > 0):
    x = x + 1
transfer(1 000 000 000 $, calyon, my-account)
print("I’m rich now")
Certification and logic in computer science

```python
x = 1
while (x > 0):
    x = x + 1
transfer(1 000 000 000 $, calyon, my-account)
print("I’m rich now")
```

$x = 2$
x = 1
while (x > 0):
    x = x + 1
transfer(1 000 000 000 $, calyon, my-account)
print("I’m rich now")

\[ x = 3 \]
x = 1
while (x > 0):
    x = x + 1
transfer(1 000 000 000 $, calyon, my-account)
print("I’m rich now")

\[ x = 4 \]
x = 1
while (x > 0):
    x = x + 1
transfer(1 000 000 000 $, calyon, my-account)
print("I’m rich now")
x = 1
while (x > 0):
    x = x + 1
transfer(1 000 000 000 $, calyon, my-account)
print("I’m rich now")

\[ x = 100 \]
x = 1
while (x > 0):
    x = x + 1
transfer(1 000 000 000 $, calyon, my-account)
print("I’m rich now")

\[ x = \ldots \]
Certification and logic in computer science

$x = 1$
while ($x > 0$):
    $x = x + 1$
transfer(1 000 000 000 $, calyon, my-account)
print("I’m rich now")
x = 1
while (x > 0):
    x = x + 1
transfer(1 000 000 000 $, calyon, my-account)
print("I’m rich now")

The core of this thesis

- **Termination** or productivity (*via* source codes)
- **Paths** to terminal states.
- For that, using **types** (data descriptors).
Certification and logic in computer science

x = 1
while (x > 0):
    x = x + 1
transfer(1 000 000 000 $, calyon, my-account)
print("I’m rich now")

The core of this thesis

- **Termination** or **productivity** *(via source codes)*
- **Paths** to terminal states.
- For that, using **types** *(data descriptors)*.

Productivity:
- O. S.
- 2, 3, 5, 7,… (primes)
Certification and logic in computer science

\[ x = 1 \]
\[ \text{while } (x > 0): \]
\[ \quad x = x + 1 \]
\[ \text{transfer(1 000 000 000 $, calyon, my-account)} \]
\[ \text{print("I’m rich now")} \]

The core of this thesis

- **Termination** or **productivity** (*via* source codes)
- **Paths** to terminal states.
- For that, using **types** (data descriptors).

**Productivity:**
- O. S.
- 2, 3, 5, 7,... (primes)

**Backtracking:**
\[ \sim \text{ Classical logic.} \]
All men are mortal. Socrates is a man. Therefore, Socrates is mortal.
All men are mortal. Socrates is a man. Therefore, Socrates is mortal.

All ringtus are delgo. Vinkri is a ringtu. Therefore, Vinkri is delgo.
All men are mortal. Socrates is a man. Therefore, Socrates is mortal.

All ringtus are delgo. Vinkri is a ringtu. Therefore, Vinkri is delgo.
All men are mortal. Socrates is a man. Therefore, Socrates is mortal.

All ringtus are delgo. Vinkri is a ringtu. Therefore, Vinkri is delgo.
All men are mortal. Socrates is a man. Therefore, Socrates is mortal.

All ringtus are delgo. Vinkri is a ringtu. Therefore, Vinkri is delgo.

\[
\forall x, \mathcal{H}(x) \Rightarrow \mathcal{M}(x) \\
\mathcal{H}(S) \Rightarrow \mathcal{M}(S) \quad \mathcal{H}(S) \\
\mathcal{M}(S)
\]
Formal logic (valar morghulis)

All men are mortal. Socrates is a man. Therefore, Socrates is mortal.

All ringtus are delgo. Vinkri is a ringtu. Therefore, Vinkri is delgo.

$$\forall x, \mathcal{H}(x) \Rightarrow \mathcal{M}(x)$$

$$\mathcal{H}(S) \Rightarrow \mathcal{M}(S) \quad \mathcal{H}(S)$$

$$\mathcal{M}(S)$$
All **men** are **mortal**. **Socrates** is a **man**. Therefore, **Socrates** is **mortal**.

All **ringtus** are **delgo**. **Vinkri** is a **ringtu**. Therefore, **Vinkri** is **delgo**.

\[
\forall x, \ H(x) \Rightarrow M(x) \\
H(S) \Rightarrow M(S) \\
\]

\[
\begin{array}{c}
H(S) \\
M(S)
\end{array}
\]
Formal logic (valar morghulis)

All men are mortal. Socrates is a man. Therefore, Socrates is mortal.

All ringtus are delgo. Vinkri is a ringtu. Therefore, Vinkri is delgo.

$\forall x, \mathcal{H}(x) \Rightarrow \mathcal{M}(x)$

$\mathcal{H}(S) \Rightarrow \mathcal{M}(S)$  \hspace{5cm} $\mathcal{H}(S)$

$\mathcal{M}(S)$

Formalization

Reduce semantic (= meaning) to mechanical/grammatical/syntactic rules.
**Turing and Computability**

**Entscheidung (1928):** given a symbolic statement, is there an *algorithmic* procedure to *decide* whether it is *true* or not?

Gödel, 1931: \( \exists \) unprovable statements

Provable \( \neq \) True

Primitive recursive functions (poor)

Can computation save mathematics?

What is an algo.?

What is effectively computable?

Turing machines (1936)

TMs are universal \( \iff \) effectively computable

\( \Rightarrow \) A prog. language \( L \) is Turing-complete if \( L \) has the same computational power as TMs.

Theorem (Turing, 1936)
The Entscheidungsproblem has a negative answer

The halting problem is undecidable: there does not exist a general method deciding whether any program terminates or not.
**Turing and Computability**

**Entscheidung (1928):** given a symbolic statement, is there an *algorithmic* procedure to *decide* whether it is *true* or not?

**Gödel, 1931:**

\[ \exists \text{ unprovable statements} \]
**Entscheidung (1928):** given a symbolic statement, is there an *algorithmic* procedure to *decide* whether it is *true* or not?

---

**Gödel, 1931:**

\[ \exists \text{ unprovable statements} \]

\[ \text{Provable } \neq \text{ True} \]
Turing and Computability

Entscheidung (1928): given a symbolic statement, is there an *algorithmic* procedure to *decide* whether it is *true* or not?

Gödel, 1931: \[ \exists \text{ unprovable statements} \]

\[
\text{Provable} \neq \text{True}
\]

primitive recursive functions (poor)
**Turing and Computability**

**Entscheidung (1928):** given a symbolic statement, is there an *algorithmic* procedure to *decide* whether it is *true* or not?

---

**Gödel, 1931:**

\[ \exists \text{ unprovable statements} \]

\[ \text{Provable } \neq \text{ True} \]

---

**primitive recursive functions (poor)**

<table>
<thead>
<tr>
<th>Can computation save mathematics?</th>
<th>Turing machines (1936)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM are <em>effectively computable</em> iff <em>implementable in a TM</em></td>
<td></td>
</tr>
<tr>
<td>( \implies ) A prog. language ( L ) is <em>Turing-complete</em> if ( L ) has the same computational power as TMs.</td>
<td></td>
</tr>
</tbody>
</table>

---

**Theorem (Turing, 1936)**

The Entscheidungsproblem has a negative answer

The halting problem is undecidable: there does not exist a general method deciding whether any program terminates or not.
Entscheidung (1928): given a symbolic statement, is there an *algorithmic* procedure to *decide* whether it is *true* or not?

Gödel, 1931: \( \exists \) unprovable statements

Provable \( \not= \) True

primitive recursive functions (poor)

Can computation save mathematics?

- What is an algo.? 
- What is *effectively* computable?
Turing and computability

**Entscheidung (1928):** given a symbolic statement, is there an *algorithmic* procedure to *decide* whether it is *true* or not?

- Gödel, 1931: \( \exists \) *unprovable* statements

Gödel, 1931: \( \exists \) unprovable statements \( \rightarrow \) Provable \( \neq \) True

- primitive recursive functions (poor)

primitive recursive functions \( \rightarrow \) Can computation save mathematics?

- What is an algo.?
- What is *effectively* computable?

**Turing machines (1936)**

TM are universal

- \( f \) *effectively computable*

  \[ \text{iff } f \text{ implementable in a } TM \]

\(~\) A prog. language \( \mathcal{L} \) is **Turing-complete**

  if \( \mathcal{L} \) has the same computational power as TMs.
Turing and computability

Entscheidung (1928): given a symbolic statement, is there an *algorithmic* procedure to *decide* whether it is *true* or not?
Turing and Computability

Entscheidung (1928): given a symbolic statement, is there an algorithmic procedure to decide whether it is true or not?

Theorem (Turing, 1936)
- The Entscheidungsproblem has a negative answer
- The halting problem is undecidable: there does not exist a general method deciding whether any program terminates or not.
Computation as rewriting

The $\lambda$-calculus

- One primitive.
- Functional paradigm.
- Turing complete.

Allows to emulate many rewriting systems e.g.:

Example (implementing natural numbers)

$O$: zero
$S$: successor

Thus:

$S\ O \simeq 1$
$S\ S\ O \simeq 2$
$S\ S\ S\ S\ S\ O \simeq 5$.

Addition

$n + O \rightarrow n$ (terminal case)

$n + S\ m \rightarrow S\ n + m$ (inductive case)

$S\ S\ S\ O + S\ S\ O \rightarrow S\ S\ S\ S\ O + S\ O \rightarrow S\ S\ S\ S\ S\ O + O \rightarrow S\ S\ S\ S\ S\ O$

$3 + 2 \rightarrow 4 + 1 \rightarrow 5 + 0 \rightarrow 5$
Computation as rewriting

The $\lambda$-calculus
- One primitive.
- **Functional** paradigm.
- Turing complete.

Allows to emulate many rewriting systems *e.g.*:

### Example (implementing natural numbers)

<table>
<thead>
<tr>
<th>$0$ : zero</th>
<th>$S$ : successor</th>
</tr>
</thead>
</table>

Thus: $S \circ 0 \simeq 1 \quad SS \circ 0 \simeq 2 \quad SSSSSSSSSSSSSSSS \circ 0 \simeq 5$. 

3 + 2 \rightarrow 4 + 1 \rightarrow 5 + 0 \rightarrow 5.
Computation as rewriting

The \( \lambda \)-calculus

- One primitive.
- Functional paradigm.
- Turing complete.

Allows to emulate many rewriting systems \( e.g. \):

**Example (implementing natural numbers)**

\[
\begin{align*}
0 & : \text{zero} & S & : \text{successor} \\
S O & \approx 1 & S S O & \approx 2 & S S S S S O & \approx 5.
\end{align*}
\]

**Addition**

\[
\begin{align*}
n + 0 & \rightarrow n \ (\text{terminal case}) & n + S m & \rightarrow S n + m \ (\text{inductive case})
\end{align*}
\]
**Computation as rewriting**

### The λ-calculus
- One primitive.
- **Functional** paradigm.
- Turing complete.

Allows to emulate many rewriting systems *e.g.*:

#### Example (implementing natural numbers)

<table>
<thead>
<tr>
<th>λ-term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>zero</td>
</tr>
<tr>
<td>$S$</td>
<td>successor</td>
</tr>
</tbody>
</table>

Thus:

- $S\ 0 \simeq 1$
- $S\ S\ 0 \simeq 2$
- $S\ S\ S\ S\ S\ 0 \simeq 5$.

#### Addition

- $n + 0 \rightarrow n$ (terminal case)
- $n + S\ m \rightarrow S\ n + m$ (inductive case)
Computation as rewriting

The \( \lambda \)-calculus

- One primitive.
- Functional paradigm.
- Turing complete.

Allows to emulate many rewriting systems \textit{e.g.}:

Example (implementing natural numbers)

<table>
<thead>
<tr>
<th>0 : zero</th>
<th>S : successor</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \circ 0 \approx 1 )</td>
<td>( S \circ S \circ 0 \approx 2 ) ( S \circ S \circ S \circ S \circ 0 \approx 5 ).</td>
</tr>
</tbody>
</table>

Addition

\( n + 0 \rightarrow n \) (terminal case) \( n + S \, m \rightarrow S \, n + m \) (inductive case)

\[
S \circ S \circ O + S \circ S \circ O \rightarrow S \circ S \circ S \circ S \circ O + S \circ O \rightarrow S \circ S \circ S \circ S \circ S \circ O + 0 \rightarrow S \circ S \circ S \circ S \circ S \circ S \circ O
\]

\( 3 + 2 \rightarrow 4 + 1 \rightarrow 5 + 0 \rightarrow 5 \)
Computation as rewriting

The $\lambda$-calculus
- One primitive.
- Functional paradigm.
- Turing complete.

Allows to emulate many rewriting systems e.g.:

Example (implementing natural numbers)

<table>
<thead>
<tr>
<th>$0$ : zero</th>
<th>$S$ : successor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thus: $S\ 0 \simeq 1$</td>
<td>$SS\ 0 \simeq 2$</td>
</tr>
<tr>
<td></td>
<td>$SSSS\ 0 \simeq 5$.</td>
</tr>
</tbody>
</table>

Addition

$n + 0 \rightarrow n$ (terminal case)  \hspace{1cm} n + Sm \rightarrow Sn + m$ (inductive case)

$SS\ 0 + SS\ 0 \rightarrow SSS\ 0 + S\ 0 \rightarrow SSSSS\ 0 + 0 \rightarrow SSSSS\ 0$

$3 + 2 \rightarrow 4 + 1 \rightarrow 5 + 0 \rightarrow 5$
Computation as rewriting

The λ-calculus

- One primitive.
- Functional paradigm.
- Turing complete.

Allows to emulate many rewriting systems e.g.:

Example (implementing natural numbers)

0: zero
S: successor

Thus:

\[ S0 \simeq 1 \]
\[ SS0 \simeq 2 \]
\[ SSSS0 \simeq 5. \]

Addition

\[ n + 0 \rightarrow n \text{ (terminal case)} \]
\[ n + Sm \rightarrow Sn + m \text{ (inductive case)} \]

\[ SSS0 + SSO \rightarrow SSSS0 + SO \rightarrow SSSSS0 + 0 \rightarrow SSSSS0 \]

3 + 2 \rightarrow 4 + 1 \rightarrow 5 + 0 \rightarrow 5
Computation as rewriting

The \(\lambda\)-calculus

- One primitive.
- **Functional** paradigm.
- Turing complete.

Allows to emulate many rewriting systems *e.g.*:

**Example (implementing natural numbers)**

\[0 : \text{zero} \quad S : \text{successor}\]

Thus:

\[S \circ 0 \simeq 1 \quad S \circ S \circ 0 \simeq 2 \quad S \circ S \circ S \circ S \circ S \circ 0 \simeq 5.\]

**Addition**

\[n + 0 \rightarrow n \text{ (terminal case)} \quad n + Sm \rightarrow Sn + m \text{ (inductive case)}\]

\[
\begin{align*}
S \circ S \circ S \circ 0 + S \circ S & \rightarrow S \circ S \circ S \circ S \circ S \circ 0 + S \circ O \\
3 + 2 & \rightarrow 4 + 1 & \rightarrow 5 + 0 & \rightarrow 5
\end{align*}
\]
Computation as rewriting

The λ-calculus

- One primitive.
- Functional paradigm.
- Turing complete.

Allows to emulate many rewriting systems e.g.:

Example (implementing natural numbers)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero</td>
</tr>
<tr>
<td>S</td>
<td>successor</td>
</tr>
</tbody>
</table>

Thus:

- \( S0 \approx 1 \)
- \( SS0 \approx 2 \)
- \( SSSS0 \approx 5 \)

Addition

- \( n + 0 \rightarrow n \) (terminal case)
- \( n + Sm \rightarrow Sn + m \) (inductive case)

\[
SSS0 + SS0 \rightarrow SSSS0 + S0 \rightarrow SSSS0 + 0 \rightarrow SSSSS0
\]

- Most structures (tabs, strings, pair of integers) can be implemented in this fashion or in the λ-calculus.
\textbf{\(\lambda\)-calculus (Church, 1928)}

Term construction (inductive grammar)

- Variable: \(x\)
- Abstraction: \(\lambda x\)
- Application: \(tu\)

Redex: \((\lambda x. r)\ s\)  
Reduct: \(r\left[\frac{s/}{x}\right]\)

Redex (reducible expression): \(\Rightarrow\) computation via substitution producing a reduct.

Non-idempotent typing operators P. Vial
**λ-calcul (Church, 1928)**

**Term construction (inductive grammar)**

- **Variable**: $x$
- **Abstraction**: $\lambda x.t$
- **Application**: $t u$

**Example:** $x(\lambda y.x y)$
\(\lambda\)-calculus (Church, 1928)

Term construction (inductive grammar)

\[ \begin{align*}
  & x \\
  & \lambda x \\
  & t \\
  & \underbrace{t \ u}_{\text{Application}} \\
  & \underbrace{x \ ((\lambda y. xy))}_{\text{Example}}
\end{align*} \]

**Redex (reducible expression):**
\( \rightsquigarrow \) computation *via* substitution
producing a **reduct**
\( \lambda \)-calculus (Church, 1928)

**Redex:**
\[(\lambda x.r)s\]

**Term construction (inductive grammar):**
- **Variable:** \(x\)
- **Abstraction:** \(\lambda x.t\)
- **Application:** \(tu\)

**Redex (reducible expression):**
\(\rightsquigarrow\) computation via substitution
producing a reduct
\( \lambda - \text{calcul (Church, 1928)} \)

**Redex:**

\( (\lambda x.r)s \)

**Term construction (inductive grammar):**

- Variable: \( x \)
- Abstraction: \( \lambda x.t \)
- Application: \( tu \)

**Redex (reducible expression):**

\( \Rightarrow \) computation via substitution

producing a reduct

Non-idempotent typing operators P. Vial

1 Presentation 6 / 46
\(\lambda\)-calculus (Church, 1928)

**Term construction (inductive grammar)**

- **Variable**: \(x\)
- **Abstraction**: \(\lambda x.t\)
- **Application**: \(t u\)

**Redex (reducible expression):**
\(\leadsto\) computation via substitution producing a **reduct**
**λ-calcul (Church, 1928)**

Reduct:
\[ r[s/x] \]

**Term construction (inductive grammar)**

- Variable: \( x \)
- Abstraction: \( \lambda x.t \)
- Application: \( tu \)

**Redex (reducible expression):**
\( \rightsquigarrow \) computation *via substitution*
producing a reduct
Let $\text{app}_2(f, x) := f(f(x))$.

- $\text{app}_2$ takes a function $f$ as an argument.
- $\text{app}_2$ is a higher-order function.
Higher-order functions and their (possible) dangers

- Let \( \text{app}_2(f, x) := f(f(x)) \).
  - \( \text{app}_2 \) takes a function \( f \) as an argument.
  - \( \text{app}_2 \) is a higher-order function.

- Autoapplication is defined by:

\[
\text{autoapp}(f) \rightarrow f(f)
\]
Higher-order functions and their (possible) dangers

- Let $\text{app}_2(f, x) := f(f(x))$.
  - $\text{app}_2$ takes a function $f$ as an argument.
  - $\text{app}_2$ is a higher-order function.

- Autoapplication is defined by:

\[
\text{autoapp}(f) \rightarrow f(f)
\]
Higher-order functions and their (possible) dangers

- Let $\text{app}_2(f, x) := f(f(x))$.
  - $\text{app}_2$ takes a function $f$ as an argument.
  - $\text{app}_2$ is a higher-order function.

- Autoapplication is defined by:
  $$\text{autoapp}(f) \rightarrow f(f)$$

- Auto-autoapplication:
  $$\text{autoapp}(\text{autoapp}) \rightarrow$$
Higher-order functions and their (possible) dangers

Let \( \text{app}_2(f, x) := f(f(x)) \).
- \( \text{app}_2 \) takes a function \( f \) as an argument.
- \( \text{app}_2 \) is a higher-order function.

Autoapplication is defined by:

\[
\text{autoapp}(f) \rightarrow f(f)
\]

Auto-autoapplication:

\[
\text{autoapp}(<\text{autoapp}>) \rightarrow \text{autoapp}(<\text{autoapp}>)
\]
Let $\text{app}_2(f, x) := f(f(x))$.
- $\text{app}_2$ takes a function $f$ as an argument.
- $\text{app}_2$ is a higher-order function.

Autoapplication is defined by:

$$\text{autoapp}(f) \rightarrow f(f)$$

Auto-autoapplication:

$$\text{autoapp}(\text{autoapp}) \rightarrow \text{autoapp}(\text{autoapp}) \rightarrow \text{autoapp}(\text{autoapp})$$
Higher-order functions and their (possible) dangers

Let $\text{app}_2(f, x) := f(f(x))$.

- $\text{app}_2$ takes a function $f$ as an argument.
- $\text{app}_2$ is a higher-order function.

Autoapplication is defined by:

$$\text{autoapp}(f) \rightarrow f(f)$$

Auto-autoapplication:

$$\text{autoapp} (\text{autoapp}) \rightarrow \text{autoapp} (\text{autoapp}) \rightarrow \text{autoapp} (\text{autoapp}) \rightarrow \text{autoapp} (\text{autoapp})$$
Higher-order functions and their (possible) dangers

- Let $\text{app}_2(f, x) := f(f(x))$.
  - $\text{app}_2$ takes a function $f$ as an argument.
  - $\text{app}_2$ is a higher-order function.

- Autoapplication is defined by:
  $$\text{autoapp}(f) \to f(f)$$

- Auto-autoapplication:
  $$\text{autoapp}(\text{autoapp}) \to \text{autoapp}(\text{autoapp}) \to \text{autoapp}(\text{autoapp})$$
  $$\to \text{autoapp}(\text{autoapp}) \to \ldots$$
Higher-order functions and their (possible) dangers

- Let \( \text{app}_2(f, x) := f(f(x)) \).
  - \( \text{app}_2 \) takes a function \( f \) as an argument.
  - \( \text{app}_2 \) is a **higher-order function**.

- Autoapplication is defined by:
  \[
  \text{autoapp}(f) \rightarrow f(f)
  \]

- Auto-autoapplication:
  \[
  \text{autoapp}(\text{autoapp}) \rightarrow \text{autoapp}(\text{autoapp}) \rightarrow \text{autoapp}(\text{autoapp})
  \rightarrow \text{autoapp}(\text{autoapp}) \rightarrow \ldots
  \]

Remember:
- Some programs that do not terminate are still meaningful: the **streams**.
- Keep on **producing** terminated values.

**Example:** The program printing 2, 3, 5, 7, 11, 13... (the list of primes).
Let \( \text{app}_2(f, x) := f(f(x)) \). 
- \( \text{app}_2 \) takes a function \( f \) as an argument.
- \( \text{app}_2 \) is a higher-order function.

Autoapplication is defined by:

\[
\text{autoapp}(f) \rightarrow f(f)
\]

Auto-autoapplication:

\[
\begin{align*}
\text{autoapp}(\text{autoapp}) & \rightarrow \text{autoapp}(\text{autoapp}) \rightarrow f(f) \\
& \rightarrow \text{autoapp}(\text{autoapp}) \rightarrow \ldots \ldots \ldots
\end{align*}
\]

Remember

- Some programs that do not terminate are still meaningful: the streams.
- Keep on producing terminated values.

Example: The program printing 2, 3, 5, 7, 11, 13... (the list of primes).

Contribution: characterizing productive streams.
Terminal states and execution/reduction strategies

\[ 2 + 3 \times 5 \rightarrow 2 + 15 \rightarrow 17 \]

Reducible (non-terminal) states

Terminal state
Let $f(x) = x \times x \times x$. What is the value of $f(3 + 4)$?
Terminal states and execution/reduction strategies

Let \( f(x) = x \times x \times x \). What is the value of \( f(3 + 4) \)?

Kim (smart)

\[
\begin{align*}
f(3 + 4) & \rightarrow f(7) \\
& \rightarrow 7 \times 7 \times 7 \\
& \rightarrow 49 \times 7 \\
& \rightarrow 343
\end{align*}
\]

Lee (not so)

\[
\begin{align*}
f(3 + 4) & \rightarrow (3 + 4) \times (3 + 4) \times (3 + 4) \\
& \rightarrow 7 \times (3 + 4) \times (3 + 4) \\
& \rightarrow 7 \times 7 \times (3 + 4) \\
& \rightarrow 7 \times 7 \times 7 \\
& \rightarrow 49 \times 7 \\
& \rightarrow 343
\end{align*}
\]

Thurston (don’t be Thurston)

\[
\begin{align*}
f(3 + 4) & \rightarrow (3 + 4) \times (3 + 4) \times (3 + 4) \\
& \rightarrow 3 \times (3 + 4) \times (3 + 4) + 4 \times (3 + 4) \times (3 + 4) \\
& \rightarrow \text{dozens of computation steps} \\
& \rightarrow 343
\end{align*}
\]
Terminal states and execution/reduction strategies

\[ 2 + 3 \times 5 \rightarrow 2 + 15 \rightarrow 17 \]

Reducible (non-terminal) states

Terminal state

Let \( f(x) = x \times x \times x \). What is the value of \( f(3 + 4) \)?

- Kim (smart)

\[ f(3 + 4) \rightarrow f(7) \rightarrow 7 \times 7 \times 7 \rightarrow 49 \times 7 \rightarrow 343 \]

- Lee (not so)

\[ f(3 + 4) \rightarrow (3 + 4) \times (3 + 4) \times (3 + 4) \rightarrow 7 \times 7 \times 7 \rightarrow 343 \]

- Thurston (don’t be Thurston)

\[ f(3 + 4) \rightarrow (3 + 4) \times (3 + 4) \times (3 + 4) \rightarrow 3 \times (3 + 4) \times (3 + 4) + 4 \times (3 + 4) \times (3 + 4) \rightarrow \text{dozens of computation steps} \rightarrow 343 \]
Terminal states and execution/reduction strategies

Non-idempotent typing operators

P. Vial

1 Presentation
Terminal states and execution/reduction strategies

Initial state \rightarrow Terminal state

Infinite path (keeps running, never reaches the terminal state)
Terminal states and execution/reduction strategies

Reduction strategy

Initial state → Terminal state

- Infinite path (keeps running, never reaches the terminal state)

Reduction strategy

- **Choice** of a reduction path.
- Can be **complete**
- Must be **certified**.
Types

**Principle**

- Types = data **descriptors**, following a **grammar**.
- Types provide certifications of **correction**.

**Example**

Let `toLetters : int → String` be the program:

- `toLetters(2) = "two"`
- `toLetters(10) = "ten"`
- `toLetters(5)`
- `toLetters("Leopard")`

Correct!

Incorrect! The arg. "Leopard" is not an int.

Non-idempotent typing operators  P. Vial  1 Presentation  9 /46
Types

Principle

- Types = data **descriptors**, following a **grammar**.
- Types provide certifications of **correction**.

**Primitive types:**

| 5: int (integer) | ”Leopard”: String (string of characters) |
## Types

### Principle

- Types = data **descriptors**, following a **grammar**.
- Types provide certifications of **correction**.

### Primitive types:

| 5: int (integer) | ”Leopard”: String (string of characters) |

### Compound types:

| length : String → int (function) |
Types

Principle

- Types = data **descriptors**, following a **grammar**.
- Types provide certifications of **correction**.
**Types**

**Principle**
- Types = data **descriptors**, following a **grammar**.
- Types provide certifications of **correction**.

**Example**

Let `toLetters : int → String` be the program:

```
toLetters(2) = "two"  toLetters(10) = "ten"
```
Types

Principle
- Types = data **descriptors**, following a **grammar**.
- Types provide certifications of **correction**.

Example
Let `toLetters : int → String` be the program:

\[
\begin{align*}
toLetters(2) &= \text{"two"} \\
toLetters(10) &= \text{"ten"}
\end{align*}
\]

\[
\begin{align*}
toLetters(5) & \quad toLetters(\text{"Leopard"})
\end{align*}
\]
**Types**

**Principle**
- Types = data **descriptors**, following a **grammar**.
- Types provide certifications of **correction**.

**Example**

Let \( \text{toLetters} : \text{int} \rightarrow \text{String} \) be the program:

\[
\begin{align*}
\text{toLetters}(2) &= \text{"two"} & \text{toLetters}(10) &= \text{"ten"}
\end{align*}
\]

\[
\begin{align*}
\text{toLetters}(5) & \quad & \text{toLetters("Leopard")}
\text{Correct!} & \quad & \text{Incorrect!} \\
\text{Returns "five"} & \quad & \text{The arg. "Leopard" is not an int.}
\end{align*}
\]
**Types**

Types = data descriptors, following a grammar. Types provide certifications of correctness.

**Primitive types:**
- `int` (integer)
- "Leopard" (string of characters)

**Compound types:**
- `length : String → int` (function)
  
  Example: `toLetters : int → String` be the program:
  
  ```
  toLetters(2) = "two"
  toLetters(10) = "ten"
  ```

  Returns "five"

  Incorrect! The arg. "Leopard" is not an int.

---

**toLetters(5)**

Correct!
Returns "five"

**toLetters("Leopard")**

Incorrect!
The arg. "Leopard" is not an int.
Types

**Types** = data descriptors, following a grammar. Types provide certifications of correctness.

**Primitive types:**
- `int` (integer)
- "Leopard" (string of characters)

**Compound types:**
- length: `String → int` (function)

**Example**
Let `toLetters : int → String` be the program:

- `toLetters(2) = "two"`
- `toLetters(10) = "ten"`
- `toLetters(5) = "five"`
- `toLetters("Leopard")` is incorrect: The arg. "Leopard" is not an int.

**Typing certificate**

```
toLetters : int → String  5 : int
```

```
toLetters(5) : String
```

Correct! Returns "five"

Incorrect! The arg. "Leopard" is not an int.
Types

Principle

Types = data descriptors, following a grammar.

Types provide certifications of correctness.

Primitive types:

- \texttt{int} (integer)
- \texttt{String} (string of characters)

Compound types:

- \texttt{length} \ : \ \texttt{String} \rightarrow \texttt{int} (function)

Example

Let \texttt{toLetters}:

\[ \texttt{toLetters : int} \rightarrow \texttt{String} \]
\[ \texttt{5 : int} \]

\[ \texttt{toLetters(5)} : \texttt{String} \]

Typing certificate

\[ \frac{\text{toLetters : int} \rightarrow \texttt{String}} {\text{toLetters(5)} : \texttt{String}} \]

Proof

\[ \frac{A \rightarrow B \quad A} {B} \]

\texttt{toLetters}:

\[ \frac{\texttt{toLetters : int} \rightarrow \texttt{String}} {\texttt{toLetters(5)} : \texttt{String}} \]

Correct!

Returns "five"

\[ \frac{\text{toLetters(5)} : \texttt{String}} {\texttt{toLetters(5)} = \text{"two"}} \]

\texttt{toLetters(10)}:

\[ \frac{\text{toLetters : int} \rightarrow \texttt{String}} {\text{toLetters(10)} : \texttt{String}} \]

Incorrect!

The arg. "Leopard" is not an int.

\[ \frac{\text{toLetters("Leopard") : String}} {\text{toLetters("Leopard") = \text{"Leopard"}}} \]
**Types**

**Types** = data descriptors, following a grammar. Types provide certifications of correctness.

**Primitive types:**
- int (integer)

**Compound types:**
- length: String → int (function)

**Example**

let toLetters: int → String be the program:

- toLetters(2) = "two"
- toLetters(10) = "ten"
- toLetters(5) = "five"
- toLetters("Leopard")

**Typing certificate**

<table>
<thead>
<tr>
<th>typing</th>
<th>proof</th>
</tr>
</thead>
<tbody>
<tr>
<td>int → String</td>
<td>String</td>
</tr>
<tr>
<td>String</td>
<td>A → B</td>
</tr>
<tr>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

```
toLetters(5)
Correct! Returns "five"
```

```
toLetters("Leopard")
Incorrect! The arg. "Leopard" is not an int.
```
Types

```
toLetters : int → String  5 : int

toLetters(5) : String

Typing certificate

A → B  A
B

Proof

This analogy goes further!
```
Types

Types = data descriptors, following a grammar. Types provide certifications of correctness.

Primitive types:
- int (integer)
- "Leopard" (string of characters)

Compound types:
- length: String → int (function)

Example:
- toLetters: int → String
  - toLetters(2) = "two"
  - toLetters(10) = "ten"
  - toLetters(5)
  - toLetters("Leopard")

Correct!

Incorrect!
The arg. "Leopard" is not an int.

Typing certificate

This analogy goes further!

Curry-Howard correspondence!
Curry-Howard (50s)

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Formula</td>
</tr>
<tr>
<td>Simply Typed Program</td>
<td>Proof</td>
</tr>
<tr>
<td>Reduction Step</td>
<td>Cut-Elimination Step</td>
</tr>
<tr>
<td>Termination</td>
<td>Termination</td>
</tr>
</tbody>
</table>

\[ \text{toLetters} : \text{int} \to \text{String} \quad \text{5 : int} \]

\[ \text{toLetters(5) : String} \]

\[ A \to B \quad A \]

\[ B \]
## Curry-Howard (50s)

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Formula</td>
</tr>
<tr>
<td>Simply Typed Program</td>
<td>Proof</td>
</tr>
<tr>
<td>Reduction Step</td>
<td>Cut-Elimination Step</td>
</tr>
<tr>
<td>Termination</td>
<td>Termination</td>
</tr>
</tbody>
</table>

### Simple types

- **Simply Typed Program**
- **Reduction Step**
- **Termination**

**Non-idempotent typing operators**

- **P. Vial**

**1 Presentation**
Curry-Howard (50s)

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Formula</td>
</tr>
<tr>
<td>Simply Typed Program</td>
<td>Proof</td>
</tr>
<tr>
<td>Reduction Step</td>
<td>Cut-Elimination Step</td>
</tr>
<tr>
<td>Termination</td>
<td>Termination</td>
</tr>
</tbody>
</table>

Simple types

- Harness higher-order comput. in a **limited** way.
- Many progs. in terminal state not typable.
### Curry-Howard (50s)

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Formula</td>
</tr>
<tr>
<td>Simply Typed Program</td>
<td>Proof</td>
</tr>
<tr>
<td>Reduction Step</td>
<td>Cut-Elimination Step</td>
</tr>
<tr>
<td>Termination</td>
<td>Termination</td>
</tr>
</tbody>
</table>

#### Simple types

- Harness higher-order comput. in a **limited** way.
- Many progs. in terminal state not typable.

#### Extensions

- Polymorphic Types
- Intersection Types
**Curry-Howard (50s)**

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Formula</td>
</tr>
<tr>
<td>Simply Typed Program</td>
<td>Proof</td>
</tr>
<tr>
<td>Reduction Step</td>
<td>Cut-Elimination Step</td>
</tr>
<tr>
<td>Termination</td>
<td>Termination</td>
</tr>
</tbody>
</table>

**Simple types**

- Harness higher-order comput. in a **limited** way.
- Many progs. in terminal state not typable.

**Polymorphic Types**

**Intersection Types**
Curry-Howard (50s)

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Formula</td>
</tr>
<tr>
<td>Simply Typed Program</td>
<td>Proof</td>
</tr>
<tr>
<td>Reduction Step</td>
<td>Cut-Elimination Step</td>
</tr>
<tr>
<td>Termination</td>
<td>Termination</td>
</tr>
</tbody>
</table>

Simple types

- Harness higher-order comput. in a **limited** way.
- Many progs. in terminal state not typable.

λ-calculus

extensions

Polymorphic Types

Intersection Types

Does not capture classical logic. Harness higher-order comput. in a **limited** way. Many progs. in terminal state not typable.

Contribution: Non-idempotent typing operators P. Vial
**Curry-Howard (50s)**

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Formula</td>
</tr>
<tr>
<td>Simply Typed Program</td>
<td>Proof</td>
</tr>
<tr>
<td>Reduction Step</td>
<td>Cut-Elimination Step</td>
</tr>
<tr>
<td>Termination</td>
<td>Termination</td>
</tr>
</tbody>
</table>

**Simple types**
- Harness higher-order comput. in a **limited** way.
- Many progs. in terminal state not typable.

**λ-calculus**
- Does not capture classical logic

**Polymorphic Types**

**Intersection Types**

**Extensions**
Curry-Howard (50s)

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Formula</td>
</tr>
<tr>
<td>Simply Typed Program</td>
<td>Proof</td>
</tr>
<tr>
<td>Reduction Step</td>
<td>Cut-Elimination Step</td>
</tr>
<tr>
<td>Termination</td>
<td>Termination</td>
</tr>
</tbody>
</table>

**Simple types**
- Harness higher-order comput. in a **limited** way.
- Many progs. in terminal state not typable.

**λ-calculus**
- Does not capture classical logic
- Get classical logic with call–cc (Griffin, 90)

**Polymorphic Types**

**Intersection Types**

**toLetters**

\[ \text{toLetters}(5) : \text{String} \]
\[ A \rightarrow B \]

Simple types

Não-idempotente: P. Vial

**Non-idempotent typing operators**
### Curry-Howard (50s)

#### Programming languages vs. Logic

<table>
<thead>
<tr>
<th>Programming languages</th>
<th>Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Formula</td>
</tr>
<tr>
<td>Simply Typed Program</td>
<td>Proof</td>
</tr>
<tr>
<td>Reduction Step</td>
<td>Cut-Elimination Step</td>
</tr>
<tr>
<td>Termination</td>
<td>Termination</td>
</tr>
</tbody>
</table>

#### Simple types
- Harness higher-order comput. in a **limited** way.
- Many progs. in terminal state not typable.

#### λ-calculus
- Does not capture classical logic

- Get classical logic with **call–cc** (Griffin, 90)

#### Contributions
- Non-idempotent typing operators
- P. Vial

---

**Presentation** 10 /46
**Cut-Elimination (animation)**

**Goal:** having a one-block proof

**Theorem (Gentzen, 1936, Prawitz, 1965)**

The cut-elimination procedure terminates (and tells us a lot of things).

**Initial proof of** $F$ (using two lemmas)

**GOAL:** having a one-block proof
**Cut-Elimination (animation)**

**Goal:**

having a one-block proof

---

Theorem (Gentzen, 1936, Prawitz, 1965)

The cut-elimination procedure terminates (and tells us a lot of things).

---

*Initial proof of* $F$ (using two lemmas)

---

*After one cut-elim. step (one lemma)*

---

**Number of occurrences:**

- $\psi$ : 6
- $\Phi$ : 1
- $\Pi$ : 1

---

**GOAL:** having a one-block proof
Goal: having a one-block proof

Number of occurrences:

\[ \psi : 10 \]
\[ \Phi : 3 \]
\[ \Pi : 1 \]
Cut-Elimination (animation)

Goal: having a one-block proof

Theorem (Gentzen, 1936, Prawitz, 1965)
The cut-elimination procedure terminates (and tells us a lot of things).

Initial proof of \( F \) (using two lemmas)

After one cut-elim. step (one lemma)

After two cut-elim. steps

Number of occurrences:
\[
\begin{align*}
\psi &: 10 \\
\Phi &: 3 \\
\Pi &: 1 \\
\end{align*}
\]

After two cut-elim. steps
**Cut-Elimination (animation)**

Theorem (Gentzen, 1936, Prawitz, 1965)

*The cut-elimination procedure terminates* (and tells us a lot of things).

Number of occurrences:

- $\psi$ : 10
- $\Phi$ : 3
- $\Pi$ : 1
**Goal**

Equivalences of the form

"the program $t$ is typable iff it can reach a terminal state"

**Idea:** several certificates to a same subprogram.
**Intersections types (Coppo, Dezani, 1980)**

**Goal**

Equivalences of the form

“*the program* $t$ *is typable iff it can reach a terminal state*”

*Proof: by the “circular” implications:*

- $t$ can reach a terminal state
- $t$ is typable
- Some reduction strategy terminates on $t$

*Idea: several certificates to a same subprogram.*
**Intersections types** (Coppo, Dezani, 1980)

**Goal**

Equivalences of the form

“the program $t$ is typable iff it can reach a terminal state”

**Idea:** several certificates to a same subprogram.

**Proof:** by the “circular” implications:

- $t$ is typable
- $t$ can reach a terminal state
- Some reduction strategy terminates on $t$

**Intersection types**

- Perhaps too expressive…
- …but certify reduction strategies!
Non-idempotency

Computation causes duplication.
Computation causes duplication.

**Non-idempotent intersection types**

Disallow duplication for typing certificates.

〜〜 Possibly many certificates for a subprogram.

〜〜 Size of certificates decreases.
**Non-idempotency**

Computation causes duplication.

**Non-idempotent intersection types**

Disallow duplication for typing certificates.

⇝ Possibly many certificates for a subprogram.

⇝ Size of certificates decreases.

Initial certificate

Initial state of the prog.

Execution
**Non-idempotency**

Computation causes duplication.

**Non-idempotent intersection types**

Disallow duplication for typing certificates.

⇝ Possibly many certificates for a subprogram.
⇝ Size of certificates decreases.

---

**Initial certificate**

**Initial state of the prog.**

---

**Execution**
Non-idempotency

Computation causes duplication.

Non-idempotent intersection types

Disallow duplication for typing certificates.

\[ \Rightarrow \] Possibly many certificates for a subprogram.

\[ \Rightarrow \] Size of certificates decreases.

---

Initial certificate

Initial state of the prog.

......

Execution
Computation causes duplication.

**Non-idempotent intersection types**

Disallow duplication for typing certificates.

⇝ Possibly many certificates for a subprogram.

⇝ Size of certificates decreases.

**Comparative (dis)advantages**

- Insanely difficult to type a particular program.
- Whole type system easier to study!
- Easier proofs of termination!
- Easier proofs of characterization!
- Easier to certify a reduction strategy!
Computation causes duplication.

**Non-idempotent intersection types**

Disallow duplication for typing certificates.

- Possibly many certificates for a subprogram.
- Size of certificates decreases.

**Initial certificate**

**Initial state of the prog.**

**Execution**

**STOP**

(cannot be reduced more)
Non-ideempotency

Computation causes duplication.

Non-idempotent intersection types
Disallow duplication for typing certificates.

⇝ Possibly many certificates for a subprogram.
⇝ Size of certificates decreases.

Initial certificate

Initial state of the prog.

......

......

STOP (cannot be reduced more)

Terminal state reached!!
Non-idempotency

Computation causes duplication.

Non-idempotent intersection types

Disallow duplication for typing certificates.

⇝ Possibly many certificates for a subprogram.

⇝ Size of certificates decreases.
Non-idempotency

Computation causes duplication.

Non-idempotent intersection types

Disallow duplication for typing certificates.

- Possibly many certificates for a subprogram.
- Size of certificates decreases.

Comparative (dis)advantages

- Insanely difficult to type a particular program.
- Whole type system easier to study!
  - Easier proofs of termination!
  - Easier proofs of characterization!
  - Easier to certify a reduction strategy!
Contents

- Gardner/de Caravalho’s non-idempotent type system.

Contribution 1:
- Quantitative types for the $\lambda\mu$-calculus (a \textit{classical} calculus)
- Certificates of reduction strategies.

Contribution 2:
- Positive answer to \textbf{Klop’s Problem}.
- Certification of an \textit{infinitary} reduction strategy. Introduction of a new type system: system $S$ (standing for \textit{sequences}).

Contribution 3:
- Around the expressive power of unconstrained infinitary intersection types.
Plan

1. Presentation

2. Non-idempotent intersection types

3. Resources for Classical Logic

4. Infinite types and productive reduction

5. Infinite types and unproductive reduction

6. Conclusion
**Head Normalization ($\lambda$)**

- **Head Normal Form**
  - Head variable ($x$)
  - Reduction strategy: reducing head redexes while it is possible.
  - $t_1$ and $t_q$ are head redexes.

- **Head Reducible Term**
  - Intersection types come to help!
  - $t_1$ and $t_q$ are head normalizing (HN) if there exists a reduction path from $t$ to a HNF.

---

Non-idempotent typing operators P. Vial

2 Non-idempotent intersection types
**Head Normalization ($\lambda$)**

- **head variable**
- **head redex**

**Head Normal Form**

**Head Reducible Term**

- $t$ is **head normalizing (HN)** if $\exists$ reduction path from $t$ to a HNF.
Head Normalization (\(\lambda\))

- \(t\) is head normalizing (HN) if \(\exists\) reduction path from \(t\) to a HNF.
- The head reduction strategy: reducing head redexes while it is possible.
Head Normalization ($\lambda$)

- $t$ is **head normalizing** (HN) if $\exists$ reduction path from $t$ to a HNF.

- The **head reduction strategy**: reducing head redexes while it is possible.
**Head Normalization (λ)**

- **t** is head normalizing (HN) if \( \exists \) reduction path from \( t \) to a HNF.

- The **head reduction strategy**: reducing head redexes while it is possible.
**Head Normalization (λ)**

- The head reduction strategy: reducing head redexes while it is possible.
**Head Normalization (\(\lambda\))**

The head reduction strategy terminates on \(t\) if \(t\) is HN (\(\exists\) path from \(t\) to a HNF).

- **The head reduction strategy**: reducing head redexes while it is possible.
A good intersection type system should enjoy:

**Subject Reduction (SR):**
Typing is stable under reduction.

**Subject Expansion (SE):**
Typing is stable under anti-reduction.

*SE is usually not verified by simple or polymorphic type systems*
A good intersection type system should enjoy:

**Subject Reduction (SR):**
Typing is stable under reduction.

**Subject Expansion (SE):**
Typing is stable under anti-reduction.

*SE is usually not verified by simple or polymorphic type systems*

\[ t \text{ can reach a terminal state} \quad \Rightarrow \quad t \text{ is typable} \quad \Rightarrow \quad \text{SR + extra arg.} \]

\[ t \text{ is typable} \quad \Rightarrow \quad \text{Some reduction strategy terminates on } t \quad \Rightarrow \quad \text{obvious} \]

typing the term. states + SE
Types are built by means of base types, arrow (→) and intersection (∧).

ACI Axioms = \[
\begin{align*}
\text{Associativity} & : (A \land D) \land C \sim A \land (D \land C) \\
\text{Commutativity} & : A \land D \sim D \land A \\
\text{Idempotence} & : A \land A \sim A
\end{align*}
\]
From Intersection Types to Quantitative Types

Types are built by means of base types, arrow ($\rightarrow$) and intersection ($\land$).

**ACI Axioms** =
\[
\begin{align*}
\text{Associativity} & : (A \land D) \land C \sim A \land (D \land C) \\
\text{Commutativity} & : A \land D \sim D \land A \\
\text{Idempotence} & : A \land A \sim A
\end{align*}
\]

<table>
<thead>
<tr>
<th>Traditional Intersection Types</th>
<th>Quantitative Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Coppo &amp; Dezani 80</strong></td>
<td><strong>Gardner 94 - Kfoury 96</strong></td>
</tr>
<tr>
<td><strong>ACI (Idempotent)</strong></td>
<td><strong>AC (Non-idempotent)</strong></td>
</tr>
<tr>
<td>Types are sets: $A \land A \land C$ is ${A, C}$</td>
<td>Types are multisets: $A \land A \land C$ is $[A, A, C]$</td>
</tr>
<tr>
<td>Qualitative properties</td>
<td>Quantitative properties</td>
</tr>
</tbody>
</table>

**Remark (non-idem. case):**
- $[A, A, C] \neq [A, C]$ i.e. $A \land A \land C \sim A \land C$.
- $[A, B] + [A] = [A, A, B]$ i.e. $\land$ is multiset sum.
Types and Rules (System $\mathcal{R}_0$)

(Strict Types) \[ \tau, \sigma := o \in \emptyset \mid I \rightarrow \tau \]

(Intersection Types) \[ I := [\sigma_i]_{i \in I} \]

Strict types $\rightsquigarrow$ syntax directed rules:

\[
\begin{align*}
\frac{\text{ax}}{x : [\tau] \vdash x : \tau} & \quad \frac{\Gamma ; x : [\sigma_i]_{i \in I} \vdash t : \tau}{\Gamma \vdash \lambda x.t : [\sigma_i]_{i \in I} \rightarrow \tau} \\
& \quad \frac{\Gamma \vdash t : [\sigma_i]_{i \in I} \rightarrow \tau \quad (\Gamma_i \vdash u : \sigma_i)_{i \in I}}{\Gamma +_{i \in I} \Gamma_i \vdash tu : \tau}
\end{align*}
\]

System $\mathcal{R}_0$

Remark

- **Relevant** system (no weakening)
- In app-rule, pointwise multiset sum \( e.g., \)

\[ (x : [\sigma] ; y : [\tau]) + (x : [\sigma, \tau]) = x : [\sigma, \sigma, \tau] ; y : [\tau] \]
Properties ($R_0$)

- **Weighted Subject Reduction**
  - Reduction preserves types and environments, and...
  - ...head reduction strictly decreases the nodes of the deriv. tree.

- **Subject Expansion**
  - Anti-reduction preserves types and environments.

**Theorem (de Carvalho)**

Let $t$ be a $\lambda$-term. Then equivalence between:

- $t$ is typable (in $R_0$)
- $t$ is $HN$
- the head reduction strategy terminates on $t$ ($\leadsto$ certification!)

**Bonus (quantitative information)**

If $\Pi$ types $t$, then $\text{size}\Pi$ bounds the number of $\text{steps}$ of the head. red. strategy on $t$. 
Let $t$ be a $\lambda$-term.

- **Head normalization (HN):** there is a path from $t$ to a head normal form.

**Nota Bene:** $\Omega$ is $\text{HNF}$ but not $\text{WN}$ ($\lambda x. y$) $\Omega$ is $\text{WN}$ but not $\text{SN}$. 

Non-idempotent typing operators P. Vial 2 Non-idempotent intersection types 21 / 46
Let $t$ be a $\lambda$-term.

- **Head normalization (HN):** there is a path from $t$ to a head normal form.

- **Weak normalization (WN):** there is at least one path from $t$ to normal form (NF).
Let $t$ be a $\lambda$-term.

- **Head normalization (HN):** there is a path from $t$ to a head normal form.

- **Weak normalization (WN):** there is at least one path from $t$ to normal form (NF).

- **Strong normalization (SN):** there is no infinite path starting at $t$. 

\[
\text{Normalization SN} \Rightarrow \text{WN} \Rightarrow \text{HN}. 
\]
Head vs Weak and Strong Normalization

Let $t$ be a $\lambda$-term.

- **Head normalization** (HN): there is a path from $t$ to a head normal form.

- **Weak normalization** (WN): there is at least one path from $t$ to normal form (NF).

- **Strong normalization** (SN): there is no infinite path starting at $t$.

Normalization

SN $\Rightarrow$ WN $\Rightarrow$ HN.

**Nota Bene:** $y \Omega$ HNF but not WN

$(\lambda x. y) \Omega$ WN but not SN
<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>HN $\mathcal{R}_0$</td>
<td>System $\mathcal{R}_0$ with any arg. can be left untyped</td>
<td>$\text{sz}(\Pi)$ bounds the number of head reduction steps</td>
</tr>
<tr>
<td>WN $\mathcal{R}_0$</td>
<td>System $\mathcal{R}_0$ with the unforgottenness criterion</td>
<td>$\text{sz}(\Pi)$ bounds the number of leftmost-outermost reduction steps (and more)</td>
</tr>
<tr>
<td>SN $\mathcal{R}_0$</td>
<td>Modify system $\mathcal{R}_0$ with the choice operator</td>
<td>$\text{sz}(\Pi)$ bounds the length of any reduction path</td>
</tr>
</tbody>
</table>

$\text{sz}(\Pi)$ bounds the number of head reduction steps.
Subject reduction and expansion in $\mathcal{R}_0$

From a typing of $(\lambda x.r)s \ldots$ to a typing of $r[s/x]$

\[
\begin{align*}
\Gamma; x:[\sigma_1, \sigma_2, \sigma_1] & \vdash r : \tau \\
\Gamma & \vdash \lambda x.r : [\sigma_1, \sigma_2, \sigma_1] \to \tau \\
\Delta_1 & \vdash s : \sigma_1 \\
\Delta_2 & \vdash s : \sigma_2 \\
\Delta_1 & \vdash s : \sigma_1
\end{align*}
\]

By relevance and non-idempotence!
Subject reduction and expansion in $R_0$

From a typing of $(\lambda x. r)s \ldots$ to a typing of $r[s/x]$

\[ \Gamma; x : [\sigma_1, \sigma_2, \sigma_1] \vdash \lambda x. r : \pi \quad \text{abs} \quad \Delta_1^a \vdash s : \sigma_1 \quad \Delta_2 \vdash s : \sigma_2 \quad \Delta_1^b \vdash s : \sigma_1 \quad \text{app} \]

\[ \Gamma + \Delta_1^a + \Delta_1^b + \Delta_2 \vdash (\lambda x. r)s : \tau \]
From a typing of \((\lambda x. r) s\) ... to a typing of \(r[s/x]\)
Subject reduction and expansion in $\mathcal{R}_0$

From a typing of $(\lambda x. r)s \ldots$ to a typing of $r[s/x]$

$$
\begin{align*}
\Gamma; \ x : [\sigma_1, \sigma_2, \sigma_1] & \vdash r : \tau \\
\Gamma & \vdash \lambda x. r : [\sigma_1, \sigma_2, \sigma_1] \rightarrow \tau \\
\Gamma + \Delta_1^a + \Delta_1^b + \Delta_2 & \vdash (\lambda x. r)s : \tau
\end{align*}
$$
Subject reduction and expansion in $\mathbb{R}_0$

From a typing of $(\lambda x. r)s \ldots$ to a typing of $r[s/x]$

By relevance and non-idempotence!

Non-idempotent typing operators \cite{Vial2002}
From a typing of \((\lambda x.r)s \ldots\) to a typing of \(r[s/x]\)
Subject reduction and expansion in $\mathcal{R}_0$

From a typing of $(\lambda x.r)s \ldots$ to a typing of $r[s/x]$
From a typing of \((\lambda x.r)s\ldots\) to a typing of \(r[s/x]\)
Subject reduction and expansion in $\mathcal{R}_0$

From a typing of $(\lambda x. r)s$ ... to a typing of $r[s/x]$

$\Pi_1^a \vdash \Delta_1^a \vdash s : \sigma_1$

$\Pi_1^b \vdash \Delta_1^b \vdash s : \sigma_1$

$\Pi_2 \vdash \Delta_2 \vdash s : \sigma_2$

$\Gamma + \Delta_1^a + \Delta_1^b + \Delta_2 \vdash r[s/x] : \tau$

Non-determinism of SR

By relevance and non-idempotence!
Subject reduction and expansion in $\mathcal{R}_0$

From a typing of $(\lambda x. r) s$ ... to a typing of $r[s/x]$

Non-determinism of SR

Non-idempotent typing operators

P. Vial

2 Non-idempotent intersection types
Plan

1. Presentation
2. Non-idempotent intersection types
3. Resources for Classical Logic
4. Infinite types and productive reduction
5. Infinite types and unproductive reduction
6. Conclusion
Intuit. logic + Peirce’s Law \(((A \to B) \to A) \to A\)
gives classical logic.
The Lambda-Mu Calculus

- Intuit. logic + Peirce’s Law \(((A \rightarrow B) \rightarrow A) \rightarrow A\)
gives classical logic.

- **Griffin 90**: `call-cc` and Felleisen’s $C$-operator typable with Peirce’s Law
  \(((A \rightarrow B) \rightarrow A) \rightarrow A\)
  \(\leadsto\) the **Curry-Howard** iso extends to classical logic
The Lambda-Mu Calculus

- Intuit. logic + Peirce’s Law \(((A \rightarrow B) \rightarrow A) \rightarrow A\)
gives classical logic.

- **Griffin 90**: call-cc and Felleisen’s C-operator typable with Peirce’s Law
  \(((A \rightarrow B) \rightarrow A) \rightarrow A\)

  \(\leadsto\) the Curry-Howard iso extends to classical logic

- **Parigot 92**: \(\lambda\mu\)-calculus = computational interpretation of classical natural deduction (e.g., vs. \(\bar{\lambda}\mu\bar{\mu}\)).
Intuit. logic + Peirce’s Law \(((A \to B) \to A) \to A\)
gives classical logic.

Griffin 90: call–cc and Felleisen’s \(C\)-operator typable with Peirce’s Law
\(((A \to B) \to A) \to A\)

\(\rightsquigarrow\) the Curry-Howard iso extends to classical logic

Parigot 92: \(\lambda\mu\)-calculus = computational interpretation of classical \textit{natural deduction} (e.g., vs. \(\hat{\lambda}\mu\hat{\mu}\)).

Captures continuations
**The $\lambda\mu$-calculus**

**Syntax:** Variables $x$ and names $\alpha$

- **(Objects)** $o ::= t \mid c$
- **(Terms)** $t, u ::= x \mid \lambda x.t \mid tu \mid \mu\alpha.c$
- **(Commands)** $c ::= [\alpha]t$

**Basic Meta-Operations:**
- $t[u/x]$ (subst.)
- $c\{u/\alpha\}$ replaces every occurrence of $[\alpha]v$ inside $t$ by $[\alpha]v u$. 

Non-idempotent typing operators P. Vial 3 Resources for Classical Logic 26 /46
The $\lambda\mu$-calculus

Syntax: Variables $x$ and names $\alpha$

(Objects) $\ o \ ::= \ t \ | \ c$
(Terms) $\ t, u \ ::= \ x \ | \ \lambda x.t \ | \ tu \ | \ \mu\alpha.c$
(Commands) $\ c \ ::= \ \lfloor\alpha\rfloor t$

Basic Meta-Operations:

- $t[u/x]$ (subst.)

Example:

- $[\alpha](x (\mu \gamma. [\alpha]x))\{u/\alpha\} =$
The \( \lambda \mu \)-calculus

Syntax: Variables \( x \) and names \( \alpha \)

- **(Objects)** \( o \) ::= \( t \mid c \)
- **(Terms)** \( t, u \) ::= \( x \mid \lambda x.t \mid tu \mid \mu \alpha.c \)
- **(Commands)** \( c \) ::= \([\alpha]t\)

Basic Meta-Operations:
- \( t[u/x] \) (subst.)
- \( c[u/\alpha] \) replaces every occurrence of \([\alpha]v\) inside \( t \) by \([\alpha]v u\).

Example:
- \([\alpha](x (\mu \gamma.[\alpha]x))\{u/\alpha\} = \)
The $\lambda\mu$-calculus

Syntax: Variables $x$ and names $\alpha$

(Objects) $o ::= t | c$

(Terms) $t, u ::= x | \lambda x.t | tu | \mu\alpha.c$

(Commands) $c ::= [\alpha]t$

Basic Meta-Operations:

- $t[u/x]$ (subst.)
- $c\{u/\alpha\}$ replaces every occurrence of $[\alpha]v$ inside $t$ by $[\alpha]v u$.

Example:

- $[\alpha](x (\mu\gamma.[\alpha]x))\{u/\alpha\} =$
The $\lambda\mu$-calculus

Syntax: Variables $x$ and names $\alpha$

(Objects) $o ::= t \mid c$

(Terms) $t, u ::= x \mid \lambda x.t \mid tu \mid \mu\alpha.c$

(Command) $c ::= [\alpha]t$

Basic Meta-Operations:

- $t[u/x]$ (subst.)

Example:

- $[\alpha](x (\mu\gamma.[\alpha]x))u \rightarrow [\alpha](x (\mu\gamma.[\alpha]x u))u$
The $\lambda\mu$-calculus

Syntax: Variables $x$ and names $\alpha$

(Objects) $o ::= t | c$

(Terms) $t, u ::= x | \lambda x. t | tu | \mu\alpha.c$

(Command) $c ::= [\alpha]t$

Basic Meta-Operations:
- $t[u/x]$ (subst.)
- $c\{u/\alpha\}$ replaces every occurrence of $[\alpha]v$ inside $t$ by $[\alpha]v u$.

Example:
- $[\alpha](x (\mu\gamma.[\alpha]x))\{u/\alpha\} = [\alpha](x (\mu\gamma.[\alpha]x u))u$
- call–cc $:= \lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x)$
### The $\lambda\mu$-calculus

**Syntax:** Variables $x$ and names $\alpha$

- (Objects) $o ::= t | c$
- (Terms) $t, u ::= x | \lambda x.t | tu | \mu\alpha.c$
- (Commands) $c ::= [\alpha]t$

**Basic Meta-Operations:**
- $t[u/x]$ (subst.)

**Example:**
- $[\alpha](x (\mu\gamma.[\alpha]x))\{u/\alpha\} = [\alpha](x (\mu\gamma.[\alpha]x u))u$
- $\text{call–cc} ::= \lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x) : ((A \rightarrow B) \rightarrow A) \rightarrow A$ (simple typing)
The $\lambda\mu$-calculus

Syntax: Variables $x$ and names $\alpha$

(Objects) \[ o ::= t \mid c \]

(Terms) \[ t, u ::= x \mid \lambda x.t \mid tu \mid \mu\alpha.c \]

(Commands) \[ c ::= [\alpha]t \]

Basic Meta-Operations:
- $t[u/x]$ (subst.)
- $c{u/\alpha}$ replaces every occurrence of $[\alpha]v$ inside $t$ by $[\alpha]v u$.

Example:
- $[\alpha](x(\mu \gamma.[\alpha]x)){u/\alpha} = [\alpha](x(\mu \gamma.[\alpha]xu))u$
- $\text{call–cc} := \lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x) : ((A \to B) \to A) \to A$ (simple typing)

Operational Semantics:
- $(\lambda x.t)u \to_\beta t[u/x]$ substitution
- $(\mu\alpha.c)u \to_\mu \mu\alpha.c{u/\alpha}$ replacement
The Typing System

Principles
Extend non-idempotent types to classical logic.

Problem 1:
finding *quantitative* descriptors suitable to classical logic

Problem 2:
guaranteeing a *decrease* in measure (weighted s.r.)
The Typing System

Principles

Extend non-idempotent types to **classical logic**.

**Problem 1:**
finding *quantitative* descriptors suitable to classical logic

\[ \leadsto \] resort to **non-idempotent union types** (below right)

**Problem 2:**
guaranteeing a *decrease* in measure (weighted s.r.)

Not obvious! The number of nodes does not work (see later).
Extend non-idempotent types to **classical logic**.

**Problem 1:**
finding *quantitative* descriptors suitable to classical logic

\[ \rightsquigarrow \text{resort to non-idempotent union types (below right)} \]

**Problem 2:**
guaranteeing a *decrease* in measure (weighted s.r.)

Not obvious! The number of nodes does not work (see later).

**Intersection:** \( \mathcal{I}, \mathcal{J} := [\mathcal{U}_k]_{k \in K} \)

\[ \mathcal{U}, \mathcal{V} =: \langle \sigma_k \rangle_{k \in K} : \text{Union} \]
The Typing System

Principles

Extend non-idempotent types to classical logic.

Problem 1:
Finding quantitative descriptors suitable to classical logic

\[ \Rightarrow \text{resort to non-idempotent union types (below right)} \]

Problem 2:
Guaranteeing a decrease in measure (weighted s.r.)

Not obvious! The number of nodes does not work (see later).

Intersection: \( \mathcal{I}, \mathcal{J} := [\mathcal{U}_k]_{k \in K} \)

\[ x : [\mathcal{U}_1, \mathcal{U}_2]; y : [\mathcal{V}] \vdash t : \mathcal{U} | \alpha : \langle \sigma_1, \sigma_2 \rangle, \beta : \langle \tau_1, \tau_2, \tau_3 \rangle \]

Union: \( \mathcal{U}, \mathcal{V} =: \langle \sigma_k \rangle_{k \in K} \)

Non-idempotent typing operators P. Vial
**The Typing System**

**Principles**

Extend non-idempotent types to **classical logic**.

---

**Problem 1:**
finding *quantitative* descriptors suitable to classical logic

\(\rightsquigarrow\) resort to **non-idempotent union types** (below right)

**Problem 2:**
guaranteeing a *decrease* in measure (weighted s.r.)

Not obvious! The number of nodes does not work (see later).

---

**Intersection:** \(I, J := [\mathcal{U}_k]_{k \in K}\)

\(x : [\mathcal{U}_1, \mathcal{U}_2]; y : [\mathcal{V}] \vdash t : \mathcal{U} \mid \alpha : \langle \sigma_1, \sigma_2 \rangle, \beta : \langle \tau_1, \tau_2, \tau_3 \rangle\)

**Union:** \(\mathcal{U}, \mathcal{V} =: \langle \sigma_k \rangle_{k \in K}: \text{Union}\)

\(A, C \text{ and non-I e.g., } \langle \sigma_1, \sigma_2 \rangle \lor \langle \sigma_1 \rangle = \langle \sigma_1, \sigma_2, \sigma_1 \rangle\)
Some Typing Rules (System $\mathcal{H}_{\lambda\mu}$)

Features

Syntax-direction, relevance, multiplicative rules accumulation of typing information.

- app-rule based upon the admissible rule of ND:

$$
\frac{A_1 \rightarrow B_1 \lor \ldots \lor A_k \rightarrow B_k \quad A_1 \land \ldots \land A_k}{B_1 \lor \ldots \lor B_k} \quad \begin{vmatrix} \text{vs.} \quad \frac{A \rightarrow B}{A} \end{vmatrix}
$$

Non-idempotent typing operators P. Vial
Some Typing Rules (System $\mathcal{H}_{\lambda\mu}$)

Features

Syntax-direction, relevance, multiplicative rules accumulation of typing information.

- **app-rule** based upon the *admissible* rule of ND:

  $A_1 \rightarrow B_1 \lor \ldots \lor A_k \rightarrow B_k \quad A_1 \land \ldots \land A_k$

  $\frac{}{B_1 \lor \ldots \lor B_k}$

  \[ \text{(vs. } \frac{A \rightarrow B}{B} \text{)} \]

- Two new rules (manipulation on the right-h.s.):

  $\Gamma \vdash t : \mathcal{U} \mid \Delta$

  $\frac{}{\Gamma \vdash [\alpha] t : \# \mid \Delta \lor \{\alpha : \mathcal{U}\}} \text{ save}$

  $\Gamma \vdash c : \# \mid \Delta$

  $\frac{}{\Gamma \vdash \mu\alpha . c : \Delta(\alpha)^* \mid \Delta \setminus \alpha} \text{ restore}$

Non-idempotent typing operators P. Vial
Some Typing Rules (System $\mathcal{H}_{\lambda\mu}$)

Features

Syntax-direction, relevance, multiplicative rules accumulation of typing information.

- **app-rule** based upon the *admissible* rule of ND:

$$
\frac{A_1 \to B_1 \lor \ldots \lor A_k \to B_k \quad A_1 \land \ldots \land A_k}{B_1 \lor \ldots \lor B_k} \quad \left(\text{vs. } \frac{A \to B \quad A}{B} \right)
$$

- Two new rules (manipulation on the right-h.s.):

$$
\frac{\Gamma \vdash t : U | \Delta}{\Gamma \vdash \lbrack \alpha \rbrack t : \# | \Delta \lor \{\alpha : U\}} \quad \text{save} \\
\frac{\Gamma \vdash c : \# | \Delta}{\Gamma \vdash \mu\alpha.c : \Delta(\alpha)^* | \Delta \setminus \alpha} \quad \text{restore}
$$

Non-idempotent typing operators P. Vial 3 Resources for Classical Logic 28/46
**Some Typing Rules (System $\mathcal{H}_{\lambda\mu}$)**

### Features

Syntax-direction, relevance, multiplicative rules **accumulation of typing information**.

- **app-rule** based upon the *admissible* rule of ND:
  
  \[
  \frac{A_1 \rightarrow B_1 \lor \ldots \lor A_k \rightarrow B_k \quad A_1 \land \ldots \land A_k}{B_1 \lor \ldots \lor B_k}
  \quad \text{(vs.} \quad \frac{A \rightarrow B \quad A}{B})
  \]

- Two new rules (manipulation on the right-h.s.):
  
  \[
  \frac{\Gamma \vdash t : U | \Delta}{\Gamma \vdash [\alpha]t : \# | \Delta \lor \{\alpha : U\}} \quad \text{save} \quad \frac{\Gamma \vdash c : \# | \Delta}{\Gamma \vdash \mu \alpha.c : \Delta(\alpha)^* | \Delta \| \alpha} \quad \text{restore}
  \]

Non-idempotent typing operators  P. Vial  3 Resources for Classical Logic  28 /46
Some Typing Rules (System $\mathcal{H}_{\lambda\mu}$)

Features

Syntax-direction, relevance, multiplicative rules accumulation of typing information.

- **app-rule** based upon the *admissible* rule of ND:

  \[
  \frac{A_1 \to B_1 \lor \ldots \lor A_k \to B_k \quad A_1 \land \ldots \land A_k}{B_1 \lor \ldots \lor B_k} \quad \text{(vs.} \frac{A \to B}{A} \text{)}
  \]

- Two new rules (manipulation on the right-h.s.):

  \[
  \frac{\Gamma \vdash t : \mathcal{U} \mid \Delta}{\Gamma \vdash [\alpha]t : \# \mid \Delta \lor \{\alpha : \mathcal{U}\}} \quad \text{save} \quad \frac{\Gamma \vdash c : \# \mid \Delta}{\Gamma \vdash \mu\alpha.c : \Delta (\alpha)^* \mid \Delta \parallel \alpha} \quad \text{restore}
  \]

Non-idempotent typing operators P. Vial 3 Resources for Classical Logic 28 /46
Some Typing Rules (System $\mathcal{H}_{\lambda\mu}$)

Features

Syntax-direction, relevance, multiplicative rules accumulation of typing information.

- **app-rule** based upon the *admissible* rule of ND:
  $$
  \frac{A_1 \rightarrow B_1 \lor \ldots \lor A_k \rightarrow B_k \quad A_1 \land \ldots \land A_k}{B_1 \lor \ldots \lor B_k}
  \quad \text{(vs. } \frac{A \rightarrow B \quad A}{B} \text{)}
  $$

- **Two new rules** (manipulation on the right-h.s.):
  $$
  \begin{align*}
  &\frac{\Gamma \vdash t : \mathcal{U} \mid \Delta}{\Gamma \vdash [\alpha]t : \# \mid \Delta \lor \{\alpha : \mathcal{U}\}} \quad \text{save} \\
  &\frac{\Gamma \vdash c : \# \mid \Delta}{\Gamma \vdash \mu\alpha.c : \Delta(\alpha)^* \mid \Delta \setminus \alpha} \quad \text{restore}
  \end{align*}
  $$
  \text{where } \_^* = \text{choice operator.}
Some Typing Rules (System $\mathcal{H}_{\lambda\mu}$)

**Features**

Syntax-direction, relevance, multiplicative rules accumulation of typing information.

- app-rule based upon the *admissible* rule of ND:

$$
\frac{A_1 \rightarrow B_1 \lor \ldots \lor A_k \rightarrow B_k \quad A_1 \land \ldots \land A_k}{B_1 \lor \ldots \lor B_k} \quad (\text{vs.} \quad \frac{A \rightarrow B}{B} \quad A)
$$

- Two new rules (manipulation on the right-h.s.):

$$
\frac{\Gamma \vdash t : \mathcal{U} \mid \Delta}{\Gamma \vdash \alpha]t : \# \mid \Delta \lor \{\alpha : \mathcal{U}\}} \quad \text{save} \quad \frac{\Gamma \vdash c : \# \mid \Delta}{\Gamma \vdash \mu\alpha.c : \Delta(\alpha)^* \mid \Delta \setminus \alpha} \quad \text{restore}
$$

where $^* = \text{choice operator}$.

$$
call/cc : [[[A \rightarrow B] \rightarrow A] \rightarrow \langle A, A \rangle \quad \text{vs.} \quad ((A \rightarrow B) \rightarrow A) \rightarrow A
$$

Non-idempotent typing operators - P. Vial

3 Resources for Classical Logic 28 /46
Properties and contributions (1)

- **Weighted Subject Reduction**
  
  with \( \text{size}(\Pi) = \begin{cases} 
  \text{number of nodes of } \Pi + \\
  \text{size of the type arities of all the names of commands +} \\
  \text{multiplicities of arguments in all the app. nodes of } \Pi. 
  \end{cases} \)

- **Subject Expansion**

---

**Theorem (Kesner, Vial, FSCD17)**

Let \( t \) be a \( \lambda \mu \)-term. Then equivalence between:

- \( t \) is typable (in \( \mathcal{H}_{\lambda \mu} \))
- \( t \) is \( \text{HN} \)
- the head reduction strategy terminates on \( t \) (thus, h.r.strat. certified!).

---

**Bonus (quantitative information)**

\( \text{size}(\Pi) \) bounds the number of steps of the head. red. strategy on \( t \).
Contributions (2)

Theorem (Kesner, Vial, FSCD17)

- System $S_{\lambda\mu}$ characterizing SN for the $\lambda\mu$-calculus.
- $sz(\Pi)$ bounds the length of any reduction sequence starting at $t$.

Extension (small-step operational semantics for the $\lambda\mu$-calculus)

- Processing substitution and replacement one occurrence at a time.
  - In $\lambda$: $(x\ y\ x\ x)[s/x] \rightsquigarrow sy\ s\ s$ (1 big step)
  - In $\lambda_{ex}$: $(x\ y\ x\ x)[s/x] \rightsquigarrow sy\ x\ x \rightsquigarrow sy\ xs \rightsquigarrow sy\ s\ s$ (3 small-steps)
- Characterization of SN (extension of $S_{\lambda\mu}$).
Plan

1. Presentation
2. Non-idempotent intersection types
3. Resources for Classical Logic
4. Infinite types and productive reduction
5. Infinite types and unproductive reduction
6. Conclusion
Klop’s Problem

- HN, WN, SN,... have been \textit{statically} characterized by various ITS.

- **Klop’s Problem:** can the set of $\infty$-WN terms be characterized by an ITS?
  
  \textit{Def:} $t$ is $\infty$-WN iff its Böhm tree does not contain $\bot$

  - **Tatsuta [07]:** an \textit{inductive} ITS cannot do it.
  - Can a \textit{coinductive} ITS characterize the set of $\infty$-WN terms?
Klop’s Problem

- HN, WN, SN,... have been *statically* characterized by various ITS.

- **Klop’s Problem**: can the set of ∞-WN terms be characterized by an ITS?
  
  *Def*: \( t \) is ∞-WN iff its Böhm tree does not contain \( \bot \)

  \[\begin{align*}
  &\quad \text{\textbf{Tatsuta [07]}: an \textit{inductive} ITS cannot do it.} \\
  &\quad \text{Can a \textit{coinductive} ITS characterize the set of ∞-WN terms?}
  \end{align*}\]

- **YES**, with ITS = *sequential* + validity criterion.
Klop’s Problem

- HN, WN, SN,... have been \textit{statically} characterized by various ITS.

- **Klop’s Problem:** can the set of $\infty$-WN terms be characterized by an ITS?
  \textit{Def:} $t$ is $\infty$-WN iff its Böhm tree does not contain $\bot$

  - \textbf{Tatsuta [07]}: an \textit{inductive} ITS cannot do it.
  - Can a \textit{coinductive} ITS characterize the set of $\infty$-WN terms?

- \textbf{YES}, with ITS = \texttt{sequential} + validity criterion.
  - But... what is infinitary normalization?
Productive vs. Unproductive Reduction

Productive reduction:

\[ \Delta f := \lambda x. f(xx) Y f \]

\[ \Delta \Delta f \]

Curry \( f \) → → → → → \( f_n(Y f) \) → → → \( \omega \)

Unproductive reduction:

\[ \Delta = \lambda x.xx, \quad \Omega = \Delta \Delta (\text{i.e. } \text{autoapp(\text{autoapp)})} \]

\[ \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \ldots \]
Productive vs. Unproductive Reduction

**Productive reduction:** \( \Delta_f := \lambda x. f(xx) \quad \text{Y}_f := \Delta_f \Delta_f \) "Curry f"

\[
\begin{align*}
\text{Y}_f & \rightarrow f(\text{Y}_f) \rightarrow f^2(\text{Y}_f) \rightarrow f^3(\text{Y}_f) \rightarrow f^4(\text{Y}_f) \rightarrow \ldots \rightarrow f^n(\text{Y}_f) \rightarrow \ldots \rightarrow \infty \ f^\omega
\end{align*}
\]
**Productive vs. Unproductive Reduction**

**Productive reduction:** \( \Delta_f := \lambda x. f(xx) \quad Y_f := \Delta_f \Delta_f \quad "Curry \ f"\)

\[ Y_f \to f(Y_f) \to f^2(Y_f) \to f^3(Y_f) \to f^4(Y_f) \to \ldots \to f^n(Y_f) \to \ldots \to \infty \ f^\omega \]
**Productive vs. Unproductive Reduction**

**Productive reduction:** $\Delta_f := \lambda x. f(xx)$ \quad $Y_f := \Delta_f \Delta_f$ "Curry $f$"

$Y_f \rightarrow f(Y_f) \rightarrow f^2(Y_f) \rightarrow f^3(Y_f) \rightarrow f^4(Y_f) \rightarrow \ldots \rightarrow f^n(Y_f) \rightarrow \ldots \rightarrow \infty f^\omega$
**Productive vs. Unproductive Reduction**

**Productive reduction:** \( \Delta_f := \lambda x. f(xx) \quad Y_f := \Delta_f \Delta_f \quad \text{”Curry } f” \)

\[
Y_f \rightarrow f(Y_f) \rightarrow f^2(Y_f) \rightarrow f^3(Y_f) \rightarrow f^4(Y_f) \rightarrow \ldots \rightarrow f^n(Y_f) \rightarrow \ldots \rightarrow \infty f^\omega
\]
**Productive vs. Unproductive Reduction**

**Productive reduction:** \( \Delta_f := \lambda x.f(xx) \quad Y_f := \Delta_f \Delta_f \quad "\text{Curry } f" \)

\[ Y_f \rightarrow f(Y_f) \rightarrow f^2(Y_f) \rightarrow f^3(Y_f) \rightarrow f^4(Y_f) \rightarrow \ldots \rightarrow f^n(Y_f) \rightarrow \ldots \rightarrow \infty \quad f^\omega \]
**Productive vs. Unproductive Reduction**

**Productive reduction:** \( \Delta_f := \lambda x. f(xx) \quad Y_f := \Delta_f \Delta_f \quad "\text{Curry } f" \)

\[ Y_f \rightarrow f(Y_f) \rightarrow f^2(Y_f) \rightarrow f^3(Y_f) \rightarrow f^4(Y_f) \rightarrow \ldots \rightarrow f^n(Y_f) \rightarrow \ldots \rightarrow \infty \ f^\omega \]
Productive reduction: $\Delta_f := \lambda x.f(xx) \quad Y_f := \Delta_f \Delta_f \quad "Curry \ f"$

$Y_f \rightarrow f(Y_f) \rightarrow f^2(Y_f) \rightarrow f^3(Y_f) \rightarrow f^4(Y_f) \rightarrow \ldots \rightarrow f^n(Y_f) \rightarrow \ldots \rightarrow \infty \ f^\omega$
Productive reduction: $\Delta_f := \lambda x.f(xx)$ \hspace{1cm} $Y_f := \Delta_f \Delta_f$ "Curry $f$"

\[ Y_f \rightarrow f(Y_f) \rightarrow f^2(Y_f) \rightarrow f^3(Y_f) \rightarrow f^4(Y_f) \rightarrow \ldots \rightarrow f^n(Y_f) \rightarrow \ldots \rightarrow \infty \ f^{\omega} \]
**Productive vs. Unproductive Reduction**

**Productive reduction:** \( \Delta_f := \lambda x.f(xx) \quad Y_f := \Delta_f \Delta_f \quad ”Curry f” \)

\[
Y_f \rightarrow f(Y_f) \rightarrow f^2(Y_f) \rightarrow f^3(Y_f) \rightarrow f^4(Y_f) \rightarrow \ldots \rightarrow f^n(Y_f) \rightarrow \ldots \rightarrow \infty \quad f^\omega
\]

Unproductive reduction: \( \Delta = \lambda x.x \circ x \), \( \Omega = \Delta \Delta \quad (i.e. \text{autoapp(autoapp)}) \)

\[
\Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \ldots
\]
**Infinite Terms**

- Infinite $\lambda$-terms.

- Infinite NF e.g., $f^\omega$.

- *Productive* reduction sequence of infinite length (*strongly converging reduction sequence*)
  $$Y_f \rightarrow f(Y_f) \ldots \text{ok} \quad \text{not } \Omega \rightarrow \Omega \ldots$$

- A term $t$ is $\infty$-WN if $\exists$ a reduction path to an $\infty$-NF.

- **Hereditary head reduction strategy:**
  from lower (root) to upper levers.
Towards Infinitary Typing

Idea

To characterize $\infty$-WN, let us unforgetfully type infinite normal forms
$\leadsto$ no part of an $\infty$-NF must be left untyped...

- Need to consider infinite derivations with a coinductive type grammar
  $(R_0 \leadsto R)$.

Problem 1: how do we perform infinite subject reduction/expansion?
Actually, this is difficult only for SE (extra-slide available)

Problem 2: the coinductive type grammar allows to define
$\rho = [\rho]_\omega \rightarrow \omega$.
Using $\rho$, we may type $\Omega$ with $\omega$ (unsound derivations)

Solution (for both problems): resort to a validity criterion called
approximability.
Towards Infinitary Typing

Idea
To characterize $\infty$-WN, let us unforgetfully type infinite normal forms $\leadsto$ no part of an $\infty$-NF must be left untyped...

- Need to consider infinite derivations with a coinductive type grammar $(R_0 \leadsto R)$.

Problem 1: how do we perform infinite subject reduction/expansion?

Actually, this is difficult only for SE (extra-slide available)

Problem 2: the coinductive type grammar allows to define $\rho = [\rho]_\omega \rightarrow o$.

Using $\rho$, we may type $\Omega$ with $o$ (unsound derivations)
Towards Infinitary Typing

Idea

To characterize $\infty$-WN, let us unforgetfully type infinite normal forms $\rightsquigarrow$ no part of an $\infty$-NF must be left untyped...

- Need to consider infinite derivations with a coinductive type grammar $(\mathcal{R}_0 \rightsquigarrow \mathcal{R})$.

**Problem 1:** how do we perform infinite subject reduction/expansion?

Actually, this is difficult only for SE (extra-slide available)

**Problem 2:** the coinductive type grammar allows to define $\rho = [\rho]_\omega \rightarrow o$.

Using $\rho$, we may type $\Omega$ with $o$ (*unsound* derivations)

- **Solution (for both problems):** resort to a *validity criterion* called approximability.
Approximability (intuitions)

- A derivation is a set of symbols, that satisfies some grammar.
- Some derivations are included in others

\[
x : \left[ \right] \rightarrow o \vdash x : \left[ \right] \rightarrow o
\]

\[
x : \left[ o \rightarrow o \right] \vdash x y : o
\]

- Informal Definition [Vial, LICS17]: a derivation \( \Pi \) is approximable if, for all finite selection of symbols \( B_0 \), there is a finite derivation \( \Pi_f \) included in \( \Pi \) and containing \( B_0 \).
**Approximability (intuitions)**

- A derivation is a set of symbols, that satisfies some grammar.
- Some derivations are included in others

\[
\begin{align*}
x : [[o] \rightarrow o] \vdash x : [o] \rightarrow o \\
y : [o] \vdash y : o
\end{align*}
\]

\[
x : [[o] \rightarrow o]; y : [o] \vdash x \cdot y : o
\]

- **Informal Definition** [Vial, LICS17]: a derivation \( \Pi \) is approximable if, for all *finite* selection of symbols \( B_0 \), there is a *finite* derivation \( \Pi_f \) included in \( \Pi \) and containing \( B_0 \).
Approximability (intuitions)

- A derivation is a set of symbols, that satisfies some grammar.
- Some derivations are included in others (black ⊆ black+red)

\[
\begin{align*}
  x : [o \rightarrow o] & \vdash x : [o] \rightarrow o \\
  y : [o] & \vdash y : o \\
  x : [o \rightarrow o] ; y : [o] & \vdash x \cdot y : o
\end{align*}
\]

- **Informal Definition [Vial, LICS17]:** a derivation \( \Pi \) is approximable if, for all finite selection of symbols \( B_0 \), there is a finite derivation \( \Pi_f \) included in \( \Pi \) and containing \( B_0 \).
**Approximability (intuitions)**

- A derivation is a set of symbols, that satisfies some grammar.
- Some derivations are included in others (black $\subseteq$ black+red)

\[
\begin{align*}
  x : [[o] \rightarrow o] & \vdash x : [o] \rightarrow o \\
  y : [o] & \vdash y : o \\
  x : [[o] \rightarrow o], y : [o] & \vdash x y : o
\end{align*}
\]

- **Informal Definition** [Vial, LICS17]: a derivation $\Pi$ is approximable if, for all finite selection of symbols $B_0$, there is a finite derivation $\Pi_f$ included in $\Pi$ and containing $B_0$. 

![Diagram of approximability]
**Approximability (intuitions)**

- A derivation is a set of symbols, that satisfies some grammar.
- Some derivations are included in others (black $\subseteq$ black+red)

\[
\begin{align*}
x : \left[ [o] \rightarrow o \right] & \vdash x : [o] \rightarrow o \\
 y : [o] & \vdash y : o \\
\hline
x : \left[ [o] \rightarrow o \right] ; y : [o] & \vdash x y : o
\end{align*}
\]

- **Informal Definition** [Vial, LICS17]: a derivation $\Pi$ is approximable if, for all finite selection of symbols $B_0$, there is a finite derivation $\Pi_f$ included in $\Pi$ and containing $B_0$. 

![Diagram showing inclusion of derivations]

Problem 3: Approximability cannot be expressed with multisets. (no tracking with multisets)

Non-idempotent typing operators P. Vial 4 Infinite types and productive reduction
**Approximability (intuitions)**

- A derivation is a set of symbols, that satisfies some grammar.
- Some derivations are included in others (black ⊆ black+red)

\[
\begin{align*}
  x : \square{[o] \rightarrow o} & \vdash x : [o] \rightarrow o \\
  y : [o] & \vdash y : o
\end{align*}
\]

\[
  x : [o] \rightarrow o ; y : [o] \vdash x y : o
\]

- **Informal Definition** [Vial, LICS17]: a derivation \( \Pi \) is approximable if, for all *finite* selection of symbols \( B_0 \), there is a *finite* derivation \( \Pi_f \) included in \( \Pi \) and containing \( B_0 \).

**Problem 3:** Approximability cannot be expressed with multisets.
(no tracking with multisets)
**Sequential Intersection**

**Solution**

Resorting to sequential intersection! (\(\rightsquigarrow\) approximability becomes definable)

- **Strict Types:**
  \[ S_k, T ::= o \in \mathcal{O} \mid (k \cdot S_k)_{k \in K} \rightarrow T \]

- **Sequence Types**  
  \((k \cdot S_k)_{k \in K}\)

- **Example:**  
  \((7 \cdot o_1, 3 \cdot o_2, 2 \cdot o_1) \rightarrow o\)

![Diagram](image)

7, 3, 2, 1 = "tracks"

- **Tracking:**  
  \((3 \cdot \sigma, 5 \cdot \tau, 9 \cdot \sigma) = (3 \cdot \sigma, 5 \cdot \tau) \sqcup (9 \cdot \tau)\)

vs.  
\([\sigma, \tau, \sigma] = [\sigma, \tau] + [\sigma]\)

Non-idempotent typing operators  
P. Vial  
4 **Infinite Types and Productive Reduction**  
37 / 46
Resorting to sequential intersection! \((\leadsto\) approximability becomes definable)

- **Strict Types:**
  \[
  S_k, T ::= o \in \mathcal{O} \mid (k \cdot S_k)_{k \in K} \rightarrow T
  \]

- **Sequence Types** \((k \cdot S_k)_{k \in K}\)

- **Example:** \((7 \cdot o_1, 3 \cdot o_2, 2 \cdot o_1) \rightarrow o\)
  
  Graph representation:

  ![Graph](image)

  \(7, 3, 2, 1 = \text{“tracks”}\)

- **Tracking:** \((3 \cdot \sigma, 5 \cdot \tau, 9 \cdot \sigma) = (3 \cdot \sigma, 5 \cdot \tau) \uplus (9 \cdot \tau)\)
  
  vs. \([\sigma, \tau, \sigma] = [\sigma, \tau] + [\sigma]\)
Resorting to sequential intersection! (~→ approximability becomes definable)

- **Strict Types:**
  \[ S_k, T ::= o \in \mathcal{O} \mid (k \cdot S_k)_{k \in K} \rightarrow T \]

- **Sequence Types**
  \[ (k \cdot S_k)_{k \in K} \]

- **Example:** \( (7 \cdot o_1, 3 \cdot o_2, 2 \cdot o_1) \rightarrow o \)

  \[
  \begin{array}{c}
  \text{7, 3, 2, 1 = "tracks"}
  \end{array}
  \]

- **Tracking:** \( (3 \cdot \sigma, 5 \cdot \tau, 9 \cdot \sigma) = (3 \cdot \sigma, 5 \cdot \tau) \sqcup (9 \cdot \tau) \)

  \[ \text{vs. } [\sigma, \tau, \sigma] = [\sigma, \tau] + [\sigma] \]
**Solution**

Resorting to sequential intersection! \( \leadsto \) approximability becomes definable

- **Strict Types:**
  \[
  S_k, T ::= o \in \mathcal{O} \mid (k \cdot S_k)_{k \in K} \to T
  \]

- **Sequence Types**
  \((k \cdot S_k)_{k \in K}\)

- **Example:** \((7 \cdot o_1, 3 \cdot o_2, 2 \cdot o_1) \to o\)

- **Tracking:**
  \((3 \cdot \sigma, 5 \cdot \tau, 9 \cdot \sigma) = (3 \cdot \sigma, 5 \cdot \tau) \uplus (9 \cdot \tau)\)

  \[\text{vs. } [\sigma, \tau, \sigma] = [\sigma, \tau] + [\sigma]\]
Resorting to sequential intersection! (~≈ approximability becomes definable)

- **Strict Types:**
  \[ S_k, T ::= \ o \in \emptyset \ | \ (k \cdot S_k)_{k \in K} \to T \]

- **Sequence Types:**
  \((k \cdot S_k)_{k \in K}\)

- **Example:**
  \((7 \cdot o_1, 3 \cdot o_2, 2 \cdot o_1) \to o\)

- **Tracking:**
  \((3 \cdot \sigma, 5 \cdot \tau, 9 \cdot \sigma) = (3 \cdot \sigma, 5 \cdot \tau) \uplus (9 \cdot \tau)\)

  vs. \([\sigma, \tau, \sigma] = [? , \tau] + [\sigma]\)

\[7, 3, 2, 1 = \text{“tracks”}\]
Derivations of $S$

$\text{ax}$ \hspace{1cm} $\text{abs}$

$x : (k \cdot T) \vdash x : T$

$C ; x : (S_k)_{k \in K} \vdash t : T$

$C, x : (S_k)_{k \in K} \vdash t : T$

$C \vdash \lambda x.t : (S_k)_{k \in K} \rightarrow T$

$C \vdash \lambda x.t : (S_k)_{k \in K} \rightarrow T$

$C \vdash t : (S_k)_{k \in K} \rightarrow T$

$(D_k \vdash u : S_k)_{k \in K}$

$C \uplus (\uplus_{k \in K} D_k) \vdash t u : T$

$\text{app}$

- System $S$ features **pointers** (called **bipositions**).

Approximability is definable in $S$

Problem 3 solved!

- Every $S$-derivation collapses on a $R$-derivation.

**Theorem**

*Given $t$, the set of the $S$-derivations typing $t$ is a complete partial order (c.p.o).*
Characterization of infinitary WN

Proposition (Vial, LICS17)

In System $S$:

- $SR$: typing is stable by productive $\infty$-reduction.
- $SE$: approximable typing stable by productive $\infty$-expansion.

Theorem (Vial, LICS17)

- A $\infty$-term $t$ is $\infty$-WN iff $t$ is unforgetfully typable by means of an approximable derivation

$\Rightarrow$ Klop’s Problem solved

- The hereditary head reduction strategy is complete for infinitary weak normalization.
**Characterization of infinitary WN**

**Proposition (Vial, LICS17)**

In System $S$:
- $SR$: typing is stable by productive $\infty$-reduction.
- $SE$: *approximable* typing stable by productive $\infty$-expansion.

**Theorem (Vial, LICS17)**

- A $\infty$-term $t$ is $\infty$-WN iff $t$ is unforgetfully typable by means of an approximable derivation

$\rightsquigarrow$ Klop’s Problem solved

- The hereditary head reduction strategy is complete for infinitary weak normalization.

**Last bonus (positive answer to TLCA Problem #20)**

System $S$ also provides a type-theoretic characterization of the **hereditary permutations** (not possible in the inductive case, Tatsuta [LICS07]).
Two questions arising from Klop’s problem

Question 1 (the set of typable terms)

What is the set of typable terms in system $R$ and $S$? (without approximability condition)

Theorem (Vial)
Every term is typable in systems $R$ and $S$ (non-trivial).
One can extract from $R$-typing the order (arity) of any $\lambda$-term.
In the infinitary relational model, no term has an empty denotation.

Question 2 (relation between $S$ and $R$)

Every $S$-derivation collapses on a $R$-derivation.
But is the converse true?

Theorem (Vial)
Every $R$-derivation is the collapse of a $S$-derivation.
One can encode any reduction choice in system $R$ b.m.o a $S$-derivation.
Two questions arising from Klop’s problem

Question 1 (the set of typable terms)
What is the set of typable terms in system $R$ and $S$? (without approximability condition)

Theorem (Vial)
- Every term is typable in systems $R$ and $S$ (non-trivial).
- One can extract from $R$-typing the order (arity) of any $\lambda$-term.
- In the infinitary relational model, no term has an empty denotation.

Question 2 (relation between $S$ and $R$)
Every $S$-derivation collapses on a $R$-derivation.
But is the converse true?
Two questions arising from Klop’s problem

Question 1 (the set of typable terms)

What is the set of typable terms in system $R$ and $S$? (without approximability condition)

Theorem (Vial)

- Every term is typable in systems $R$ and $S$ (non-trivial).
- One can extract from $R$-typing the order (arity) of any $\lambda$-term.
- In the infinitary relational model, no term has an empty denotation.

Question 2 (relation between $S$ and $R$)

Every $S$-derivation collapses on a $R$-derivation. But is the converse true?

Theorem (Vial)

- Every $R$-derivation is the collapse of a $S$-derivation.
- One can encode any reduction choice in system $R$ b.m.o. a $S$-derivation.
**Difficulties**

- In the *productive* cases (HN, WN, SN, $\infty$-WN), in i.t.s., one types the normal forms and uses subject expansion.
  
  \[
  \text{normalizing terms} \subseteq \text{typable terms}
  \]

- Here, no form of productivity/stabilization.

- We develop a corpus of methods inspired by **first order model theory** (last part of the dissertation).
Plan

1 Presentation

2 Non-idempotent intersection types

3 Resources for Classical Logic

4 Infinite types and productive reduction

5 Infinite types and unproductive reduction

6 Conclusion
## Beyond this thesis

**Intersection types via Grothendieck construction**

[Mazza,Pellissier,Vial, POPL2018]

- Categorical generalization of ITS. à la Melliès-Zeilberger.
- Type systems = 2-operads (see below).

### Type systems as 2-operads

- **Level 1:** $\Gamma \vdash t : B$ \hspace{1cm} $t = \text{multimorphism}$ from $\Gamma$ to $B$.
- **Level 2:** if $\Gamma \vdash t : B \xrightarrow{SR} \Gamma \vdash t' : B$, \hspace{1cm} $t \rightsquigarrow t' = \text{2-morphism}$ from $t$ to $t'$.

### Construction of an i.t.s. via a Grothendieck construction (pullbacks).

- **Modularity:** retrieving automatically  
  *e.g., e.g.*, Coppo-Dezani, Gardner, $\mathcal{H}_0$, call-by-value + $\mathcal{H}_{\lambda\mu}$ (use cyclic 2-operads)
The $\lambda\mu$-calculus:

- Characterization of HN and SN with non-idempotent/quantitative methods (extension of $\mathcal{R}_0$).
- Certification of reduction strategies.
- **Upper** bounds on normalizing strategies.
- Small-step operational semantics and SN (extension).
What we did and what we shall do

The $\lambda\mu$-calculus:
- Characterization of HN and SN with non-idempotent/quantitative methods (extension of $R_0$).
- Certification of reduction strategies.
- **Upper** bounds on normalizing strategies.
- Small-step operational semantics and SN (extension).

Perspectives
- **Exact** bounds on normalizing strategies (à la Bernadet-Lengrand).
- Quantitative types for other classical calculi (e.g., Curien-Herbelin’s $\check{\lambda}\check{\mu}\check{\tilde{\mu}}$).
- Studying the model underlying $H_{\lambda\mu}$. 
What we did and what we shall do

The $\lambda\mu$-calculus: Characterization of HN and SN with non-idempotent/quantitative methods (extension of $R_0$).
Certification of reduction strategies.
Upper bounds on normalizing strategies.
Small-step operational semantics and SN (extension).

Perspectives
Exact bounds on normalizing strategies (à la Bernadet-Lengrand).
Quantitative types for other classical calculi (e.g., Curien-Herbelin’s $\bar{\lambda}\mu$).
Studying the model underlying $H\lambda\mu$.

Klop’s Problem and Infinitary Normalization
Characterizing infinitary weak normalization.
Certifying an infinitary reduction strategy (HHN).
Positive answer to TLCA Problem # 20.
Introduction of system $S$ (sequential intersection, non-idem. flavor).
Introduction of a validity criterion (approximability).

Perspectives
Other forms of $\infty$-normalization (other calculi, $\infty$-SN).
Relations between system $S$ and ludics, GoI, indexed LL.

Relations with Grellois-Mellies infinitary model of LL.
Non-idempotent typing operators P. Vial
What we did and what we shall do

Klop’s Problem and Infinitary Normalization

- Characterizing *infinitary* weak normalization.
- Certifying an *infinitary* reduction strategy (HHN).
- Positive answer to TLCA Problem # 20.

- Introduction of system $\mathbb{S}$ (sequential intersection, non-idem. flavor).
- Introduction of a validity criterion (*approximability*).
Klop’s Problem and Infinitary Normalization
- Characterizing *infinitary* weak normalization.
- Certifying an *infinitary* reduction strategy (HHN).
- Positive answer to TLCA Problem # 20.
- Introduction of system $S$ (sequential intersection, non-idem. flavor).
- Introduction of a validity criterion (*approximability*).

Perspectives
- Other forms of $\infty$-normalization (other calculi, $\infty$-SN)
- Relations between system $S$ and ludics, GoI, indexed LL...
- Relations with Grellois-Melliès infinitary model of LL.
Thank you

Thank you for your attention!
Peirce’s Law in Classical Natural Deduction

\[
\begin{align*}
(A \rightarrow B) \rightarrow A & \vdash (A \rightarrow B) \rightarrow A & A \vdash A, B \\
\vdash A \rightarrow B, A & & \vdash A \rightarrow B, A \\
(A \rightarrow B) \rightarrow A & \vdash A, A \\
\vdash (A \rightarrow B) \rightarrow A & \vdash A \\
\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A & & \text{Standard Style}
\end{align*}
\]
Peirce’s Law in Classical Natural Deduction

\[(A \rightarrow B) \rightarrow A \vdash (A \rightarrow B) \rightarrow A\]

\[\vdash A \rightarrow B, A\]

\[\vdash (A \rightarrow B) \rightarrow A \vdash A, A\]

\[\vdash (A \rightarrow B) \rightarrow A\]

\[\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A\]

Standard Style
Peirce’s Law in Classical Natural Deduction

\[
\begin{align*}
(A \to B) \to A & \vdash (A \to B) \to A & \frac{A \vdash A, B}{\vdash A \to B, A} \\
(A \to B) \to A & \vdash A, A & \frac{(A \to B) \to A \vdash A, A}{\vdash ((A \to B) \to A) \to A} \end{align*}
\]

**Standard Style**
Peirce’s Law in Classical Natural Deduction

\[
\frac{(A \rightarrow B) \rightarrow A \vdash (A \rightarrow B) \rightarrow A}{(A \rightarrow B) \rightarrow A \vdash A, A}
\]

\[
\frac{A \vdash A, B}{\vdash A \rightarrow B, A}
\]

\[
\frac{(A \rightarrow B) \rightarrow A \vdash A, A}{(A \rightarrow B) \rightarrow A \vdash A}
\]

\[
\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A
\]

Standard Style
**Peirce’s Law in Classical Natural Deduction**

\[
\frac{(A \to B) \to A \vdash (A \to B) \to A}{(A \to B) \to A \vdash A, A}
\]

\[
\frac{(A \to B) \to A \vdash A, A}{(A \to B) \vdash A}
\]

\[
\vdash ((A \to B) \to A) \to A
\]

**Standard Style**

\[
\frac{A \vdash A | B}{A \vdash B | A}^{\text{act}}
\]

\[
\frac{(A \to B) \to A \vdash (A \to B) \to A | A}{(A \to B) \to A \vdash A | A}
\]

\[
\frac{(A \to B) \to A \vdash A | A}{(A \to B) \to A \vdash A | A}
\]

\[
\vdash ((A \to B) \to A) \to A | A
\]

**Focussed Style**
Peirce’s Law in Classical Natural Deduction

\[
\begin{align*}
(A \rightarrow B) \rightarrow A &\vdash (A \rightarrow B) \rightarrow A & \quad A \vdash A, B \\
(A \rightarrow B) \rightarrow A &\vdash A, A \\
(A \rightarrow B) &\vdash A \\
\vdots &\vdots \\
\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A
\end{align*}
\]

Standard Style

\[
\begin{align*}
A \vdash A | B \\
A \vdash B | A
\end{align*}
\]

\[
\begin{align*}
(A \rightarrow B) \rightarrow A &\vdash (A \rightarrow B) \rightarrow A | A \\
(A \rightarrow B) &\vdash A | A \\
\vdots &\vdots \\
\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A | A
\end{align*}
\]

Focussed Style
Peirce’s Law in Classical Natural Deduction

\[(A \to B) \to A \vdash (A \to B) \to A\]
\[A \vdash A, B\]
\[\vdash A \to B, A\]
\[(A \to B) \to A \vdash A, A\]
\[\vdash (A \to B) \to A\]
\[\vdash ((A \to B) \to A) \to A\]

Standard Style

<table>
<thead>
<tr>
<th>Act</th>
<th>Non-idempotent typing operators P. Vial</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Conclusion</td>
<td>47</td>
</tr>
</tbody>
</table>
### Typing call–cc in $\mathcal{H}_{\lambda\mu}$

<table>
<thead>
<tr>
<th>$y : [[[A \rightarrow B] \rightarrow A] \vdash y : [[A \rightarrow B] \rightarrow A]$</th>
<th>$x : [A] \vdash x : A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x : [A] \vdash [\alpha]x : #$</td>
<td>$\alpha : A$</td>
</tr>
<tr>
<td>$x : [A] \vdash \mu\beta.[\alpha]x : B$</td>
<td>$\alpha : A$</td>
</tr>
<tr>
<td>$\vdash \lambda x.\mu\beta.[\alpha]x : [A] \rightarrow B$</td>
<td>$\alpha : A$</td>
</tr>
<tr>
<td>$y : [[[A \rightarrow B] \rightarrow A] \vdash y(\lambda x.\mu\beta.[\alpha]x) : A$</td>
<td>$\alpha : A$</td>
</tr>
<tr>
<td>$y : [[[A \rightarrow B] \rightarrow A] \vdash [\alpha]y(\lambda x.\mu\beta.[\alpha]x) : #$</td>
<td>$\alpha : \langle A, A \rangle$</td>
</tr>
<tr>
<td>$y : [[[A \rightarrow B] \rightarrow A] \vdash \mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x) : \langle A, A \rangle$</td>
<td></td>
</tr>
<tr>
<td>$\vdash \lambda y.\mu\alpha.[\alpha]y(\lambda x.\mu\beta.[\alpha]x) : [[[A \rightarrow B] \rightarrow A] \rightarrow \langle A, A \rangle$</td>
<td></td>
</tr>
</tbody>
</table>

Non-idempotent typing operators

P. Vial

6 Conclusion
Typing call–cc in $\mathcal{H}_{\lambda\mu}$

\[ y : [[[[A] \rightarrow B] \rightarrow A] \vdash y : [[A] \rightarrow B] \rightarrow A \]

\[ x : [A] \vdash x : A \]

\[ x : [A] \vdash [\alpha]x : \# \mid \alpha : A \]

\[ x : [A] \vdash \mu \beta.[\alpha]x : B \mid \alpha : A \]

\[ \vdash \lambda x.\mu \beta.[\alpha]x : [A] \rightarrow B \mid \alpha : A \]

\[ y : [[[A] \rightarrow B] \rightarrow A] \vdash y : [[A] \rightarrow B] \rightarrow A \]

\[ \vdash \lambda x.\mu \beta.[\alpha]x : [A] \rightarrow B \mid \alpha : A \]

\[ y : [[[A] \rightarrow B] \rightarrow A] \vdash y(\lambda x.\mu \beta.[\alpha]x) : A \mid \alpha : A \]

\[ y : [[[A] \rightarrow B] \rightarrow A] \vdash [\alpha]y(\lambda x.\mu \beta.[\alpha]x) : \# \mid \alpha : \langle A, A \rangle \]

\[ y : [[[A] \rightarrow B] \rightarrow A] \vdash \mu \alpha.[\alpha]y(\lambda x.\mu \beta.[\alpha]x) : \langle A, A \rangle \]

\[ \vdash \lambda y.\mu \alpha.[\alpha]y(\lambda x.\mu \beta.[\alpha]x) : [[[A] \rightarrow B] \rightarrow A] \rightarrow \langle A, A \rangle \]
Let $A$ be any formula. We then set $R_A := ((\ldots \to A) \to A) \to A$ i.e. $R_A = R_A \to A$. 

\[
\begin{array}{c}
  \frac{R_A \vdash R_A}{\vdash R_A \to A} \\
  \frac{R_A \vdash R_A}{\vdash R_A \to A}
\end{array} \quad \begin{array}{c}
  \frac{R_A \vdash R_A}{\vdash R_A \to A} \\
  \frac{R_A \vdash R_A}{\vdash R_A}
\end{array}
\]
Let $A$ be any formula.
We then set $R_A := (((\ldots) \to A) \to A) \to A$ i.e. $R_A = R_A \to A$. 

\[
\begin{array}{c}
R_A \vdash R_A \text{ i.e. } R_A \to A \\
\hline
R_A \vdash A \\
\hline
\vdash R_A \to A \text{ i.e. } R_A \\
\hline
\vdash A
\end{array}
\]
Let $A$ be any formula.
We then set $R_A := (((\ldots) \to A) \to A) \to A$ i.e. $R_A = R_A \to A$.

$$
\begin{array}{ccc}
R_A \vdash R_A & & R_A \vdash R_A \\
\hline
& R_A \vdash A & \\
\hline
& \vdash R_A \to A & \\
\hline
& & \vdash A \\
\end{array}
$$

$$
\begin{array}{ccc}
R_A \vdash R_A & & R_A \vdash R_A \\
\hline
& R_A \vdash A & \\
\hline
& \vdash R_A & \\
\hline
& & \vdash R_A \\
\end{array}
$$

Non-idempotent typing operators

P. Vial

6 Conclusion
Infinite formulas are unsound

Let $A$ be any formula.

We then set $R_A := (((\ldots) \to A) \to A) \to A$ i.e. $R_A = R_A \to A$.

\[
\begin{align*}
\frac{x : R_A \vdash x : R_A}{\vdash \lambda x.xxx : R_A \to A} & \quad \frac{x : R_A \vdash x : R_A}{\vdash \lambda x.xxx : R_A \to A} & \quad \frac{R_A \vdash R_A}{\vdash \lambda x.xxx : R_A} \\
\frac{R_A \vdash R_A}{\vdash \Omega : A}
\end{align*}
\]
$\Pi' \triangleright \Gamma \vdash f^\omega : o$

Every Variable is Typed

\[ \Gamma = f : [[o] \to o]_\omega \text{ (infinite multiplicity)} \]
\[ \Pi' \triangleright f : [[o] \to o]_\omega \vdash f^\omega : o \text{ can be truncated into } \Pi'_4 \]
Truncation (Figures)

\[ \Pi' \triangleright f : [\omega \rightarrow \omega] \vdash f^\omega : o \text{ can be truncated into } \Pi'_4 \]

Non-idempotent typing operators P. Vial

Conclusion
$\Pi' \triangleright f : [[o] \to o]_\omega \vdash f^\omega : o$ can be truncated into $\Pi'_4$
Truncation (Figures)

\[ \Pi' \triangleright f : \left[ [o] \rightarrow o \right]_\omega \vdash f^\omega : o \text{ can be truncated into } \Pi'_3 \]
$\Pi' \triangleright f : [[o] \rightarrow o]_\omega \vdash f^\omega : o$ can be truncated into $\Pi'_3$

![Diagram showing the truncation process](image)
\( f^\omega \) may be replaced by \( f^3(\Delta_f \Delta_f) \) in \( \Pi'_3 \), yielding \( \Pi^3_3 \):

\[
\begin{array}{c}
[ ] \rightarrow o \\
[0] \rightarrow o \\
[0] \rightarrow o \\
[ ] \rightarrow o
\end{array}
\]
$f^\omega$ may be replaced by $f^3(\Delta_f \Delta_f)$ in $\Pi'_3$, yielding $\Pi_3^3$:
\( \Pi_3^3 \) may be expanded 3 times, yielding \( \Pi_3 \triangleright \Delta_f \Delta_f \):

![Diagram showing the expansion process](image-url)
Back to $\Pi'_4$, level 4 truncation of $\Pi'$:
Truncation (Figures)

$f^\omega$ may be replaced by $f^4(\Delta_f \Delta_f)$ in $\Pi'_3$, yielding $\Pi_4^4$:

Non-idempotent typing operators P. Vial

Conclusion
$f^\omega$ may be replaced by $f^4(\Delta_f \Delta_f)$ in $\Pi'_3$, yielding $\Pi^4_4$:
$\Pi_4^4$ may be expanded 4 times, yielding $\Pi_4 \triangleright \Delta_f \Delta_f$:

\[
\begin{array}{c}
[ ] \rightarrow o \\
[o] \rightarrow o \\
[o] \rightarrow o \\
[o] \rightarrow o
\end{array}
\]
Initial derivation (inf. term, inf. deriv.)
Truncation (Figures)

- Initial derivation (inf. term, inf. deriv.)
- Truncation (inf t., f. deriv.)
Truncation (Figures)

Initial derivation (inf. term, inf. deriv.)

Truncation (inf t., f. deriv.)

Subject subst. (fin. t., fin. d.)
Truncation (Figures)

- Initial derivation (inf. term, inf. deriv.)
- Truncation (inf t., f. deriv.)
- Subject subst. (fin. t., fin. d.)
- Expansion (∼\(Y_f\) is typed)
**Truncation (Figures)**

- Initial derivation (inf. term, inf. deriv.)
- Truncation (inf t., f. deriv.)
- Subject subst. (fin. t., fin. d.)
- Expansion (\( \rightsquigarrow Y_f \) is typed)
- Take the join for all trunc.

[Diagram showing the relationships between the terms and derivations, with nodes labeled \( \Delta_f \Delta_f \), \( f \), and \( \@ \), and arrows indicating the flow of derivation and truncation.]
Support candidates

What is a correct type?

Support:
\{\varepsilon, 1, 4, 4 \cdot 1, 4 \cdot 3, 4 \cdot 8\}
What is a correct type?

Support candidates:

Wrong Labels

Support:
\{\varepsilon, 1, 4, 4\cdot1, 4\cdot3, 4\cdot8\}
**Support candidates**

What is a correct type?

**Correct Labels**

**Support:**
\{\varepsilon, 1, 4, 4\cdot1, 4\cdot3, 4\cdot8\}

**Type:** \((4 \cdot (8 \cdot o_3, 3 \cdot o_1) \rightarrow o_2) \rightarrow o_1\)
What is a correct type?

Support candidates:
- a set of positions that is the support of a type

\[ \begin{align*}
\varepsilon, & \ 1, \ 4, \ 4 \cdot 1, \ 4 \cdot 3, \ 4 \cdot 8 \\
\varepsilon, & \ 1, \ 4, \ 4 \cdot 3
\end{align*} \]
What is a correct type?

**Support:**
\{\varepsilon, 1, 4, 4\cdot1, 4\cdot3, 4\cdot8\}

**Wrong Support**

**Support:**
\{\varepsilon, 1, 4, 4\cdot3\}

Support candidate: a set of positions that is the support of a type
Support candidates

What is a correct type?

\[ \{ \varepsilon, 1, 4, 4 \cdot 1, 4 \cdot 3, 4 \cdot 8 \} \]

\[ \{ \varepsilon, 1, 4, 4 \cdot 3 \} \]

Support candidate: a set of positions that is the support of a type

- \( c \cdot k \rightarrow_{t_1} c \) (a candidate supp is a tree)
- \( c \cdot k \rightarrow_{t_2} c \cdot 1 \) (if a node does not have a 1-child, it is a leaf)
**Support candidates**

What is a correct type?

Support:
\{\varepsilon, 1, 4, 4\cdot1, 4\cdot3, 4\cdot8\}

Support:
\{\varepsilon, 1, 4, 4\cdot3\}

**Support candidate:** a set of positions that is the support of a type
- \(c \cdot k \rightarrow_{t_1} c\) (a candidate supp is a tree)
- \(c \cdot k \rightarrow_{t_2} c \cdot 1\) (if a node does not have a 1-child, it is a leaf)

**Lemma:** Let \(C \subseteq \mathbb{N}^*\). Then \(\exists T\) type, \(C = \text{supp}(T)\) iff \(C \neq \emptyset\) and \(C\) stable under \(\rightarrow_{t_1}, \rightarrow_{t_2}\).
We want to show that every term $t$ is typable in $S$. 
We want to show that every term $t$ is typable in $S$.

**Idea:** we try to capture the notion of **bisupport candidate**: a set of pointers that is the bisupport of a $S$-derivation typing $t$ and have a proposition of the form:

**Proposition:** let $t$ be a term and $B$ a set of bipositions. Then, 
$\exists P$ derivation, $B = \text{bisupp}(P)$ iff $B \neq \emptyset$ and $B$ stable under $\rightarrow_1$, $\rightarrow_2$, $\rightarrow_3$, ... [see Prop. 12.3, p. 260]
Bisupport Candidates

We want to show that every term $t$ is typable in $S$.

*Idea:* we try to capture the notion of **bisupport candidate**: a set of pointers that is the bisupport of a $S$-derivation typing $t$ and have a proposition of the form:

**Proposition:** let $t$ be a term and $B$ a set of bipositions. Then, $\exists P$ derivation, $B = \text{bisupp}(P)$ iff $B \neq \emptyset$ and $B$ stable under $\rightarrow_1, \rightarrow_2, \rightarrow_3, \ldots$ [see Prop. 12.3, p. 260]

We must find suitable stability conditions.
Bisupport Candidates

- We want to show that every term \( t \) is typable in \( S \).

- **Idea:** we try to capture the notion of **bisupport candidate**: a set of pointers that is the bisupport of a \( S \)-derivation typing \( t \) and have a proposition of the form:

  **Proposition:** let \( t \) be a term and \( B \) a set of bipositions. Then, \( \exists P \) derivation, \( B = \text{bisupp}(P) \) iff \( B \neq \emptyset \) and \( B \) stable under \( \rightarrow_1, \rightarrow_2, \rightarrow_3, \ldots \) [see Prop. 12.3, p. 260]

- We must find suitable stability conditions.

- Then, we show that there is actually a *non-empty* set that satisfies them.
Guidelines of the proof

- Reduce the problem (“every term is S-typable”) to a parametrized first order theory $\mathcal{T}_t \ (t \in \Lambda)$.

- Establish a “completeness-like” property:

  **Prop.**: let $t \in \Lambda$. Then $t$ is S-typable iff $\mathcal{T}_t$ is consistent.

- How do we prove that $\mathcal{T}_t$ cannot be contradictory?

Remark: works for the infinitary $\lambda$-calculus!
Guidelines of the proof

- Reduce the problem (“every term is $S$-typable”) to a parametrized first order theory $T_t$ ($t \in \Lambda$).

- Establish a “completeness-like” property:

  Prop.: let $t \in \Lambda$. Then $t$ is $S$-typable iff $T_t$ is consistent.

- How do we prove that $T_t$ cannot be contradictory?
  1. Assume *ad absurdum* that $T_t$ is contradictory for some $t$. Then, there is a finite proof $C$ (standing for chain) that $T_t$ is contradictory.
  2. If $C$ “visits” redexes, $C$ is not decypherable. But we cannot eliminate redexes in all generality (*e.g.*, in mute terms). What can we do?

Non-idempotent typing operators

P. Vial

6 Conclusion
Guidelines of the proof

- Reduce the problem (“every term is S-typable”) to a parametrized first order theory $\mathcal{T}_t$ ($t \in \Lambda$).

- Establish a “completeness-like” property:

  Prop.: let $t \in \Lambda$. Then $t$ is S-typable iff $\mathcal{T}_t$ is consistent.

- How do we prove that $\mathcal{T}_t$ cannot be contradictory?
  1. Assume *ad absurdum* that $\mathcal{T}_t$ is contradictory for some $t$. Then, there is a finite proof $\mathcal{C}$ (standing for chain) that $\mathcal{T}_t$ is contradictory.
  2. If $\mathcal{C}$ “visits” redexes, $\mathcal{C}$ is not decypherable. But we cannot eliminate redexes in all generality (*e.g.*, in mute terms). What can we do?

  - **Fundamental idea:** There is a finite reduction strategy (called the **collapsing strategy**) $t \rightarrow t'$ such that $\mathcal{C}$ can be *residuated* into a chain $\mathcal{C}'$ of $t'$ that does not interact with redex ($\mathcal{C}'$ is called a **normal chain**).
  - We prove that $\mathcal{C}'$ cannot exist. So $\mathcal{C}$ does not either *i.e.* there is not proof of contradiction.
  - Thus, $\mathcal{T}_t$ is consistent!
Guidelines of the Proof

- Reduce the problem ("every term is $S$-typable") to a parametrized first order theory $\mathcal{T}_t$ ($t \in \Lambda$).

- Establish a "completeness-like" property:

  Prop.: let $t \in \Lambda$. Then $t$ is $S$-typable iff $\mathcal{T}_t$ is consistent.

- How do we prove that $\mathcal{T}_t$ cannot be contradictory?

  1. Assume ad absurdum that $\mathcal{T}_t$ is contradictory for some $t$. Then, there is a finite proof $C$ (standing for chain) that $\mathcal{T}_t$ is contradictory.

  2. If $C$ "visits" redexes, $C$ is not decypherable. But we cannot eliminate redexes in all generality (e.g., in mute terms). What can we do?

  3. Fundamental idea: There is a finite reduction strategy (called the collapsing strategy) $t \rightarrow t'$ such that $C$ can be residuated into a chain $C'$ of $t'$ that does not interact with redex ($C'$ is called a normal chain).

  4. We prove that $C'$ cannot exist. So $C$ does not either i.e. there is not proof of contradiction.

  5. Thus, $\mathcal{T}_t$ is consistent!

- Remark: works for the infinitary $\lambda$-calculus!
**Theorem (complete unsoundness):** in $\mathcal{R}$, every term is typable.

[Th 12.1, p. 276]

**Conclusion**
Theorem (complete unsoundness): in $\mathcal{R}$, every term is typable.
[Th 12.1, p. 276]

Theorem: if $t$ is a zero-term, then, $t$ is typable with $o$.
[Th 12.2, p. 276]
**Order**

**Theorem (complete unsoundness):** in $\mathcal{K}$, every term is typable.
[Th 12.1, p. 276]

**Theorem:** if $t$ is a zero-term, then, $t$ is typable with $o$.
[Th 12.2, p. 276]

**Definition (relational model):** For all closed $\lambda$-term $t$, we set

$$[[t]] = \{\tau \mid \vdash t : \tau \text{ is derivable}\}$$
**Theorem (complete unsoundness):** in $\mathcal{R}$, every term is typable.
[Th 12.1, p. 276]

**Theorem:** if $t$ is a zero-term, then, $t$ is typable with $o$.
[Th 12.2, p. 276]

**Definition (relational model):** For all closed $\lambda$-term $t$, we set

$$[[t]] = \{ \tau \mid \vdash t : \tau \text{ is derivable} \}$$

**Corollary:** This yields a *non-sensible* model that discriminates terms according to their order:

if $t$ and $u$ are two terms of different orders, then $[[t]] \neq [[u]]$.

**First model to do this!**