Garbage Collection for a calculus with indices

Luis Francisco Ziliani

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Licenciatura's Thesis, 2009

Outline

Why study λ-calculus related theories

Invented by A. Church in the '30. Why study it?

- λ-calculus is the base of functional programming and theorem provers.
- Simple and yet powerful.
- But its naïve implementation is not useful in all cases, and there might be no obvious extension for what is needed.
- For the Automath theorem prover, de Bruijn created a way to unique identify variables by the use of *indices*.
- Also, the need for performance lead to *explicit substitutions*.

Why study λ-calculus related theories

Invented by A. Church in the '30. Why study it?

- λ-calculus is the base of functional programming and theorem provers.
- Simple and yet powerful.
- But its naïve implementation is not useful in all cases, and there might be no obvious extension for what is needed.
- For the Automath theorem prover, de Bruijn created a way to unique identify variables by the use of *indices*.
- Also, the need for performance lead to *explicit substitutions*.

Why study λ-calculus related theories

Invented by A. Church in the '30. Why study it?

- λ-calculus is the base of functional programming and theorem provers.
- Simple and yet powerful.
- But its naïve implementation is not useful in all cases, and there might be no obvious extension for what is needed.
- For the Automath theorem prover, de Bruijn created a way to unique identify variables by the use of *indices*.
- Also, the need for performance lead to *explicit substitutions*.

Why study λ-calculus related theories

Invented by A. Church in the '30. Why study it?

- λ-calculus is the base of functional programming and theorem provers.
- Simple and yet powerful.
- But its naïve implementation is not useful in all cases, and there might be no obvious extension for what is needed.
- For the Automath theorem prover, de Bruijn created a way to unique identify variables by the use of *indices*.
- Also, the need for performance lead to *explicit substitutions*.

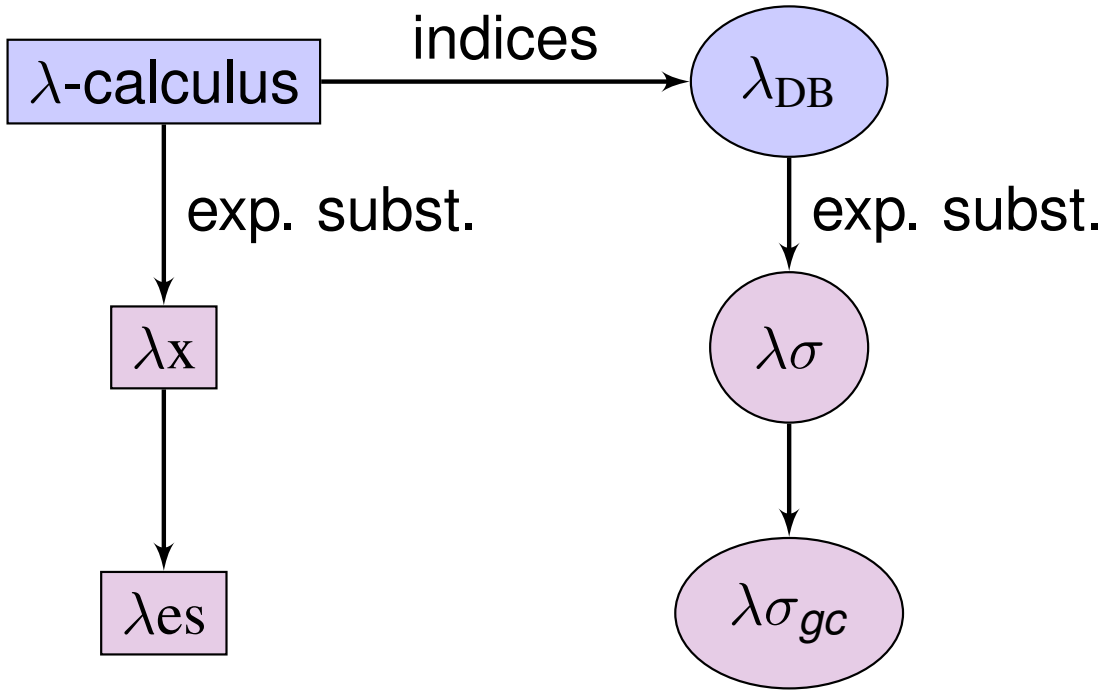
Why study λ-calculus related theories

Invented by A. Church in the '30. Why study it?

- λ-calculus is the base of functional programming and theorem provers.
- Simple and yet powerful.
- But its naïve implementation is not useful in all cases, and there might be no obvious extension for what is needed.
- For the Automath theorem prover, de Bruijn created a way to unique identify variables by the use of *indices*.
- Also, the need for performance lead to *explicit substitutions*.

Part of the λ-calculus hierarchy

What we are going to see



Part of the λ-calculus hierarchy

What we are going to see

λ-calculus

The *F* programming language

- The terms of *F* are constructed by
  - Naturals $(0, 1, \ldots)$ with built-in operations $+, \times, \ldots, >, <, =, \ldots$
  - Booleans (*true*, *false*) with built-in operations *if* $\cdot$ *then* $\cdot$ *else* $\cdot, \wedge, \vee, \ldots$
  - Variables $x, y, \ldots$
  - Functions (for instance, $f(x) = x + 1$). In this case x is the *abstracted* variable of f and $x + 1$ is the body.
  - Application of terms to terms (for instance, $f\ 3$). Here, 3 is the *argument* of f. Application is left-associative: $f\ 2\ 1$ is $(f\ 2)\ 1$.
- *One computation step* ($\rightarrow$) is defined to an application $f\ T$ as the act to replace each occurrence of the abstracted variable in the body with the argument. For example, $f\ 3 \rightarrow 3 + 1 = 4$

The *F* programming language

- The terms of *F* are constructed by
  - Naturals $(0, 1, \ldots)$ with built-in operations $+, \times, \ldots, >, <, =, \ldots$
  - Booleans (*true*, *false*) with built-in operations *if* $\cdot$ *then* $\cdot$ *else* $\cdot, \wedge, \vee, \ldots$
  - Variables $x, y, \ldots$
  - Functions (for instance, $f(x) = x + 1$). In this case x is the *abstracted* variable of f and $x + 1$ is the body.
  - Application of terms to terms (for instance, $f\ 3$). Here, 3 is the *argument* of f. Application is left-associative: $f\ 2\ 1$ is $(f\ 2)\ 1$.
- *One computation step* ($\rightarrow$) is defined to an application $f\ T$ as the act to replace each occurrence of the abstracted variable in the body with the argument. For example, $f\ 3 \rightarrow 3 + 1 = 4$

The *F* programming language

- The terms of *F* are constructed by
  - Naturals $(0, 1, \ldots)$ with built-in operations $+, \times, \ldots, >, <, =, \ldots$
  - Booleans (*true*, *false*) with built-in operations *if* $\cdot$ *then* $\cdot$ *else* $\cdot, \wedge, \vee, \ldots$
  - Variables $x, y, \ldots$
  - Functions (for instance, $f(x) = x + 1$). In this case x is the *abstracted* variable of f and $x + 1$ is the body.
  - Application of terms to terms (for instance, $f\ 3$). Here, 3 is the *argument* of f. Application is left-associative: $f\ 2\ 1$ is $(f\ 2)\ 1$.
- *One computation step* ($\rightarrow$) is defined to an application $f\ T$ as the act to replace each occurrence of the abstracted variable in the body with the argument. For example, $f\ 3 \rightarrow 3 + 1 = 4$

The *F* programming language

- The terms of *F* are constructed by
  - Naturals $(0, 1, \ldots)$ with built-in operations $+, \times, \ldots, >, <, =, \ldots$
  - Booleans (*true*, *false*) with built-in operations *if* $\cdot$ *then* $\cdot$ *else* $\cdot, \wedge, \vee, \ldots$
  - Variables $x, y, \ldots$
  - Functions (for instance, $f(x) = x + 1$). In this case x is the *abstracted* variable of f and $x + 1$ is the body.
  - Application of terms to terms (for instance, $f\ 3$). Here, 3 is the *argument* of f. Application is left-associative: $f\ 2\ 1$ is $(f\ 2)\ 1$.
- *One computation step* ($\rightarrow$) is defined to an application $f\ T$ as the act to replace each occurrence of the abstracted variable in the body with the argument. For example, $f\ 3 \rightarrow 3 + 1 = 4$

The *F* programming language

- The terms of *F* are constructed by
  - Naturals $(0, 1, \ldots)$ with built-in operations $+, \times, \ldots, >, <, =, \ldots$
  - Booleans (*true*, *false*) with built-in operations *if* $\cdot$ *then* $\cdot$ *else* $\cdot, \wedge, \vee, \ldots$
  - Variables $x, y, \ldots$
  - Functions (for instance, $f(x) = x + 1$). In this case x is the *abstracted* variable of f and $x + 1$ is the body.
  - Application of terms to terms (for instance, $f\ 3$). Here, 3 is the *argument* of f. Application is left-associative: $f\ 2\ 1$ is $(f\ 2)\ 1$.
- *One computation step* ($\rightarrow$) is defined to an application $f\ T$ as the act to replace each occurrence of the abstracted variable in the body with the argument. For example, $f\ 3 \rightarrow 3 + 1 = 4$

The *F* programming language

- The terms of *F* are constructed by
  - Naturals $(0, 1, \ldots)$ with built-in operations $+, \times, \ldots, >, <, =, \ldots$
  - Booleans (*true*, *false*) with built-in operations *if* $\cdot$ *then* $\cdot$ *else* $\cdot, \wedge, \vee, \ldots$
  - Variables $x, y, \ldots$
  - Functions (for instance, $f(x) = x + 1$). In this case x is the *abstracted* variable of f and $x + 1$ is the body.
  - Application of terms to terms (for instance, $f\ 3$). Here, 3 is the *argument* of f. Application is left-associative: $f\ 2\ 1$ is $(f\ 2)\ 1$.
- *One computation step* ($\rightarrow$) is defined to an application $f\ T$ as the act to replace each occurrence of the abstracted variable in the body with the argument. For example, $f\ 3 \rightarrow 3 + 1 = 4$

The *F* programming language

- The terms of *F* are constructed by
  - Naturals $(0, 1, \ldots)$ with built-in operations $+, \times, \ldots, >, <, =, \ldots$
  - Booleans (*true*, *false*) with built-in operations *if* $\cdot$ *then* $\cdot$ *else* $\cdot, \wedge, \vee, \ldots$
  - Variables $x, y, \ldots$
  - Functions (for instance, $f(x) = x + 1$). In this case x is the *abstracted* variable of f and $x + 1$ is the body.
  - Application of terms to terms (for instance, $f\ 3$). Here, 3 is the *argument* of f. Application is left-associative: $f\ 2\ 1$ is $(f\ 2)\ 1$.
- *One computation step* ($\rightarrow$) is defined to an application $f\ T$ as the act to replace each occurrence of the abstracted variable in the body with the argument. For example, $f\ 3 \rightarrow 3 + 1 = 4$

The *F* programming language

Examples

$$fc(x) = \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1$$

$$fc\ 3 \rightarrow \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots$$

$$\rightarrow 3 \times 2 \times 1 = 6$$

$$f(x) = x + 1 \qquad h(y) = y\ 4$$

$$g(x) = x \times 2 \qquad o(x) = x\ x$$

$$h\ f \rightarrow f\ 4 \rightarrow 4 + 1 = 5$$

$$h\ g \rightarrow g\ 4 \rightarrow 4 \times 2 = 8$$

$$h\ h \rightarrow h\ 4 \rightarrow 4\ 4 = ???$$

$$o\ o \rightarrow o\ o \rightarrow o\ o \rightarrow \ldots$$

The *F* programming language

Examples

$$\begin{aligned} fc(x) &= \textit{if}\ (x > 1)\ \textit{then}\ x \times fc\ (x-1)\ \textit{else}\ 1 \\ fc\ 3 &\rightarrow \textit{if}\ (3 > 1)\ \textit{then}\ 3 \times fc\ (3-1)\ \textit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\ &\rightarrow 3 \times 2 \times 1 = 6 \end{aligned}$$

$$\begin{aligned} f(x) &= x + 1 \qquad\qquad h(y) = y\ 4 \\ g(x) &= x \times 2 \qquad\qquad o(x) = x\ x \end{aligned}$$

$$\begin{aligned} h\ f &\rightarrow f\ 4 \rightarrow 4 + 1 = 5 \\ h\ g &\rightarrow g\ 4 \rightarrow 4 \times 2 = 8 \\ h\ h &\rightarrow h\ 4 \rightarrow 4\ 4 = ??? \\ o\ o &\rightarrow o\ o \rightarrow o\ o \rightarrow \ldots \end{aligned}$$

The *F* programming language

Examples

$$\begin{aligned} fc(x) &= \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\ fc\ 3 &\rightarrow \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\ &\rightarrow 3 \times 2 \times 1 = 6 \end{aligned}$$

$$\begin{aligned} f(x) &= x + 1 & \qquad h(y) &= y\ 4 \\ g(x) &= x \times 2 & \qquad o(x) &= x\ x \end{aligned}$$

$$\begin{aligned} h\ f &\rightarrow f\ 4 \rightarrow 4 + 1 = 5 \\ h\ g &\rightarrow g\ 4 \rightarrow 4 \times 2 = 8 \\ h\ h &\rightarrow h\ 4 \rightarrow 4\ 4 = ??? \\ o\ o &\rightarrow o\ o \rightarrow o\ o \rightarrow \ldots \end{aligned}$$

The *F* programming language

Examples

$$
\begin{array}{rcl}
fc(x) & = & \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\
fc\ 3 & \rightarrow & \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\
 & \rightarrow & 3 \times 2 \times 1 = 6
\end{array}
$$

$$
\begin{array}{rcl}
f(x) & = & x + 1 \\
g(x) & = & x \times 2
\end{array}
\qquad
\begin{array}{rcl}
h(y) & = & y\ 4 \\
o(x) & = & x\ x
\end{array}
$$

$$
\begin{array}{rcl}
h\ f & \rightarrow & f\ 4 \rightarrow 4 + 1 = 5 \\
h\ g & \rightarrow & g\ 4 \rightarrow 4 \times 2 = 8 \\
h\ h & \rightarrow & h\ 4 \rightarrow 4\ 4 = ??? \\
o\ o & \rightarrow & o\ o \rightarrow o\ o \rightarrow \ldots
\end{array}
$$

The *F* programming language

Examples

$$\begin{aligned} fc(x) &= \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\ fc\ 3 &\rightarrow \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\ &\rightarrow 3 \times 2 \times 1 = 6 \end{aligned}$$

$$\begin{aligned} f(x) &= x + 1 \\ g(x) &= x \times 2 \end{aligned} \qquad \begin{aligned} h(y) &= y\ 4 \\ o(x) &= x\ x \end{aligned}$$

$$\begin{aligned} h\ f &\rightarrow f\ 4 \rightarrow 4 + 1 = 5 \\ h\ g &\rightarrow g\ 4 \rightarrow 4 \times 2 = 8 \\ h\ h &\rightarrow h\ 4 \rightarrow 4\ 4 = ??? \\ o\ o &\rightarrow o\ o \rightarrow o\ o \rightarrow \ldots \end{aligned}$$

The *F* programming language

Examples

$$\begin{aligned} fc(x) &= \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\ fc\ 3 &\rightarrow \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\ &\rightarrow 3 \times 2 \times 1 = 6 \end{aligned}$$

$$\begin{aligned} f(x) &= x + 1 & \qquad h(y) &= y\ 4 \\ g(x) &= x \times 2 & \qquad o(x) &= x\ x \end{aligned}$$

$$\begin{aligned} h\ f &\rightarrow f\ 4 \rightarrow 4 + 1 = 5 \\ h\ g &\rightarrow g\ 4 \rightarrow 4 \times 2 = 8 \\ h\ h &\rightarrow h\ 4 \rightarrow 4\ 4 = ??? \\ o\ o &\rightarrow o\ o \rightarrow o\ o \rightarrow \ldots \end{aligned}$$

The *F* programming language

Examples

$$\begin{aligned} fc(x) &= \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\ fc\ 3 &\rightarrow \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\ &\rightarrow 3 \times 2 \times 1 = 6 \end{aligned}$$

$$\begin{aligned} f(x) &= x + 1 & \qquad h(y) &= y\ 4 \\ g(x) &= x \times 2 & \qquad o(x) &= x\ x \end{aligned}$$

$$\begin{aligned} h\ f &\rightarrow f\ 4 \rightarrow 4 + 1 = 5 \\ h\ g &\rightarrow g\ 4 \rightarrow 4 \times 2 = 8 \\ h\ h &\rightarrow h\ 4 \rightarrow 4\ 4 = ??? \\ o\ o &\rightarrow o\ o \rightarrow o\ o \rightarrow \ldots \end{aligned}$$

The *F* programming language

Examples

$$
\begin{array}{rcl}
fc(x) & = & \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\
fc\ 3 & \rightarrow & \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\
 & \rightarrow & 3 \times 2 \times 1 = 6
\end{array}
$$

$$
\begin{array}{rcl}
f(x) & = & x + 1 \\
g(x) & = & x \times 2
\end{array}
\qquad\qquad
\begin{array}{rcl}
h(y) & = & y\ 4 \\
o(x) & = & x\ x
\end{array}
$$

$$
\begin{array}{rcl}
h\ f & \rightarrow & f\ 4 \rightarrow 4 + 1 = 5 \\
h\ g & \rightarrow & g\ 4 \rightarrow 4 \times 2 = 8 \\
h\ h & \rightarrow & h\ 4 \rightarrow 4\ 4 = ??? \\
o\ o & \rightarrow & o\ o \rightarrow o\ o \rightarrow \ldots
\end{array}
$$

The *F* programming language

Examples

$$
\begin{aligned}
fc(x) &= \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\
fc\ 3 &\rightarrow \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\
&\rightarrow 3 \times 2 \times 1 = 6
\end{aligned}
$$

$$
\begin{aligned}
f(x) &= x + 1 & \qquad h(y) &= y\ 4 \\
g(x) &= x \times 2 & \qquad o(x) &= x\ x
\end{aligned}
$$

$$
\begin{aligned}
h\ f &\rightarrow f\ 4 \rightarrow 4 + 1 = 5 \\
h\ g &\rightarrow g\ 4 \rightarrow 4 \times 2 = 8 \\
h\ h &\rightarrow h\ 4 \rightarrow 4\ 4 = ??? \\
o\ o &\rightarrow o\ o \rightarrow o\ o \rightarrow \ldots
\end{aligned}
$$

The *F* programming language

Examples

$$
\begin{aligned}
fc(x) &= \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\
fc\ 3 &\rightarrow \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\
&\rightarrow 3 \times 2 \times 1 = 6
\end{aligned}
$$

$$
\begin{aligned}
f(x) &= x + 1 & \qquad h(y) &= y\ 4 \\
g(x) &= x \times 2 & \qquad o(x) &= x\ x
\end{aligned}
$$

$$
\begin{aligned}
h\ f &\rightarrow f\ 4 \rightarrow 4 + 1 = 5 \\
h\ g &\rightarrow g\ 4 \rightarrow 4 \times 2 = 8 \\
h\ h &\rightarrow h\ 4 \rightarrow 4\ 4 = ??? \\
o\ o &\rightarrow o\ o \rightarrow o\ o \rightarrow \ldots
\end{aligned}
$$

The *F* programming language

Examples

$$
\begin{aligned}
fc(x) &= \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\
fc\ 3 &\rightarrow \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\
&\rightarrow 3 \times 2 \times 1 = 6
\end{aligned}
$$

$$
\begin{aligned}
f(x) &= x + 1 & \qquad h(y) &= y\ 4 \\
g(x) &= x \times 2 & \qquad o(x) &= x\ x
\end{aligned}
$$

$$
\begin{aligned}
h\ f &\rightarrow f\ 4 \rightarrow 4 + 1 = 5 \\
h\ g &\rightarrow g\ 4 \rightarrow 4 \times 2 = 8 \\
h\ h &\rightarrow h\ 4 \rightarrow 4\ 4 = ??? \\
o\ o &\rightarrow o\ o \rightarrow o\ o \rightarrow \ldots
\end{aligned}
$$

The *F* programming language

Examples

$$
\begin{aligned}
fc(x) &= \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\
fc\ 3 &\rightarrow \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\
&\rightarrow 3 \times 2 \times 1 = 6
\end{aligned}
$$

$$
\begin{aligned}
f(x) &= x + 1 & \qquad h(y) &= y\ 4 \\
g(x) &= x \times 2 & \qquad o(x) &= x\ x
\end{aligned}
$$

$$
\begin{aligned}
h\ f &\rightarrow f\ 4 \rightarrow 4 + 1 = 5 \\
h\ g &\rightarrow g\ 4 \rightarrow 4 \times 2 = 8 \\
h\ h &\rightarrow h\ 4 \rightarrow 4\ 4 = ??? \\
o\ o &\rightarrow o\ o \rightarrow o\ o \rightarrow \ldots
\end{aligned}
$$

The *F* programming language

Examples

$$
\begin{array}{rcl}
fc(x) & = & \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\
fc\ 3 & \rightarrow & \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\
 & \rightarrow & 3 \times 2 \times 1 = 6
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}rcl}
f(x) & = & x + 1 & h(y) & = & y\ 4 \\
g(x) & = & x \times 2 & o(x) & = & x\ x
\end{array}
$$

$$
\begin{array}{rcl}
h\ f & \rightarrow & f\ 4 \rightarrow 4 + 1 = 5 \\
h\ g & \rightarrow & g\ 4 \rightarrow 4 \times 2 = 8 \\
h\ h & \rightarrow & h\ 4 \rightarrow 4\ 4 = ??? \\
o\ o & \rightarrow & o\ o \rightarrow o\ o \rightarrow \ldots
\end{array}
$$

The *F* programming language

Examples

$$
\begin{array}{rcl}
fc(x) & = & \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\
fc\ 3 & \rightarrow & \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\
 & \rightarrow & 3 \times 2 \times 1 = 6
\end{array}
$$

$$
\begin{array}{rcl}
f(x) & = & x + 1 \\
g(x) & = & x \times 2
\end{array}
\qquad\qquad
\begin{array}{rcl}
h(y) & = & y\ 4 \\
o(x) & = & x\ x
\end{array}
$$

$$
\begin{array}{rcl}
h\ f & \rightarrow & f\ 4 \rightarrow 4 + 1 = 5 \\
h\ g & \rightarrow & g\ 4 \rightarrow 4 \times 2 = 8 \\
h\ h & \rightarrow & h\ 4 \rightarrow 4\ 4 = ??? \\
o\ o & \rightarrow & o\ o \rightarrow o\ o \rightarrow \ldots
\end{array}
$$

The *F* programming language

Examples

$$
\begin{aligned}
fc(x) &= \mathit{if}\ (x > 1)\ \mathit{then}\ x \times fc\ (x-1)\ \mathit{else}\ 1 \\
fc\ 3 &\rightarrow \mathit{if}\ (3 > 1)\ \mathit{then}\ 3 \times fc\ (3-1)\ \mathit{else}\ 1 = 3 \times fc\ 2 \rightarrow \ldots \\
&\rightarrow 3 \times 2 \times 1 = 6
\end{aligned}
$$

$$
\begin{aligned}
f(x) &= x + 1 & \qquad h(y) &= y\ 4 \\
g(x) &= x \times 2 & \qquad o(x) &= x\ x
\end{aligned}
$$

$$
\begin{aligned}
h\ f &\rightarrow f\ 4 \rightarrow 4 + 1 = 5 \\
h\ g &\rightarrow g\ 4 \rightarrow 4 \times 2 = 8 \\
h\ h &\rightarrow h\ 4 \rightarrow 4\ 4 = ??? \\
o\ o &\rightarrow o\ o \rightarrow o\ o \rightarrow \ldots
\end{aligned}
$$

From F to λ-calculus

Pure λ-calculus is similar to F, but

- It does not have constants (i.e. naturals and booleans).
  How do we do arithmetic?
- Functions are represented as "inline" anonymous functions.
  - In F the definition and the application of a function are different things:
    Definition: $f(x) = T$. Application: $f\ U$.
  - In λ-calculus both concepts are joined.
    $f\ U$ is written as $(\lambda x.T)\ U$.
    $f\ (f\ U)$ is written as $(\lambda x.T)\ ((\lambda x.T)\ U)$.
    How can we do a recursive call?

From F to λ-calculus

Pure λ-calculus is similar to F, but

- It does not have constants (i.e. naturals and booleans).
  How do we do arithmetic?
- Functions are represented as "inline" anonymous functions.
  - In F the definition and the application of a function are different things:
    Definition: $f(x) = T$. Application: $f\ U$.
  - In λ-calculus both concepts are joined.
    $f\ U$ is written as $(\lambda x.T)\ U$.
    $f\ (f\ U)$ is written as $(\lambda x.T)\ ((\lambda x.T)\ U)$.
    How can we do a recursive call?

From F to λ-calculus

Pure λ-calculus is similar to F, but

- It does not have constants (i.e. naturals and booleans).
  How do we do arithmetic?
- Functions are represented as "inline" anonymous functions.
  - In F the definition and the application of a function are different things:
    Definition: $f(x) = T$. Application: $f\ U$.
  - In λ-calculus both concepts are joined.
    $f\ U$ is written as $(\lambda x.T)\ U$.
    $f\ (f\ U)$ is written as $(\lambda x.T)\ ((\lambda x.T)\ U)$.
    How can we do a recursive call?

From F to λ-calculus

Pure λ-calculus is similar to F, but

- It does not have constants (i.e. naturals and booleans).
  How do we do arithmetic?
- Functions are represented as "inline" anonymous functions.
  - In F the definition and the application of a function are different things:
    Definition: $f(x) = T$. Application: $f\ U$.
  - In λ-calculus both concepts are joined.
    $f\ U$ is written as $(\lambda x.T)\ U$.
    $f\ (f\ U)$ is written as $(\lambda x.T)\ ((\lambda x.T)\ U)$.
    How can we do a recursive call?

From F to λ-calculus

Pure λ-calculus is similar to F, but

- It does not have constants (i.e. naturals and booleans).
  How do we do arithmetic?
- Functions are represented as "inline" anonymous functions.
  - In F the definition and the application of a function are different things:
    Definition: $f(x) = T$. Application: $f\ U$.
  - In λ-calculus both concepts are joined.
    $f\ U$ is written as $(\lambda x.T)\ U$.
    $f\ (f\ U)$ is written as $(\lambda x.T)\ ((\lambda x.T)\ U)$.
    How can we do a recursive call?

Formalization

The computation step, called β-reduction, is defined as

$$(\lambda x.T)\ U \quad \rightarrow_\beta \quad T[\![x/U]\!]$$

And the substitution is defined as

$$\begin{array}{rcll} x[\![x/T]\!] & = & T & \\ y[\![x/T]\!] & = & y & (x \neq y) \\ (T_1\ T_2)[\![x/U]\!] & = & T_1[\![x/U]\!]\ T_2[\![x/U]\!] & \\ (\lambda y.T)[\![x/U]\!] & = & \lambda y.(T[\![x/U]\!]) & \end{array}$$

We will use the convention that every abstracted variable is different.

An introduction on λ-calculus .

Example.

$$(\lambda x.(\lambda y.x))\ T \rightarrow_\beta (\lambda y.x)[\![x/T]\!] = (\lambda y.T)$$

Variables can be

- *bound* to a λ, like x in $(\lambda x.x\ y)$.
- or *free*, like y in the same example.

An introduction on λ-calculus .

Example.

$$(\lambda x.(\lambda y.x))\ T \ \rightarrow_\beta \ (\lambda y.x)[\![x/T]\!] \ = \ (\lambda y.T)$$

Variables can be

- *bound* to a λ, like x in $(\lambda x.x\ y)$.
- or *free*, like y in the same example.

An introduction on λ-calculus .

Example.

$$(\lambda x.(\lambda y.x))\ T \ \rightarrow_\beta \ (\lambda y.x)[\![x/T]\!] \ = \ (\lambda y.T)$$

Variables can be

- *bound* to a λ, like x in $(\lambda x.x\ y)$.
- or *free*, like y in the same example.

An introduction on λ-calculus .

Example.

$$(\lambda x.(\lambda y.x))\ T \ \rightarrow_\beta \ (\lambda y.x)[\![x/T]\!] \ = \ (\lambda y.T)$$

Variables can be

- *bound* to a λ, like x in $(\lambda x.x\ y)$.
- or *free*, like y in the same example.

α-congruence

Are these two terms different?

$$(\lambda x.x) \qquad\qquad (\lambda y.y)$$

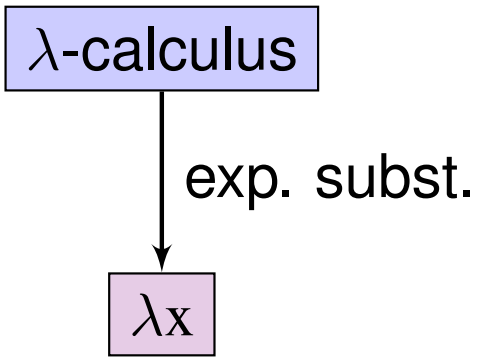
Syntactically, yes, but they behave the same way.

We will not make that distinction, by use of the

α-congruence: Two terms are α-congruent if they only differ in the *bounded variables*.

Part of the λ-calculus hierarchy

What we are going to see



Explicit substitutions

- In traditional λ-calculus the substitution is made all at once.

$$(\lambda x.(\lambda y.x)\ x)T \ \rightarrow_\beta \ ((\lambda y.x)\ x)[\![x/T]\!] = ((\lambda y.T)\ T)$$

- This can lead to a size explosion when having multiple copies of a variable.
- In practice, we need more control over what's going on during the substitution.

$$\begin{aligned}(\lambda x.(\lambda y.x)\ x)T &\rightarrow_\beta ((\lambda y.x)\ x)[\![x/T]\!] \\ &\rightarrow_\beta (x[\![y/x]\!])[\![x/T]\!] \\ &\rightarrow x[\![x/T]\!] \\ &\rightarrow T\end{aligned}$$

Explicit substitutions

- In traditional λ-calculus the substitution is made all at once.

$$(\lambda x.(\lambda y.x)\ x)T \ \rightarrow_\beta \ ((\lambda y.x)\ x)[\![x/T]\!] = ((\lambda y.T)\ T)$$

- This can lead to a size explosion when having multiple copies of a variable.
- In practice, we need more control over what's going on during the substitution.

$$\begin{aligned}(\lambda x.(\lambda y.x)\ x)T &\rightarrow_\beta ((\lambda y.x)\ x)[\![x/T]\!] \\ &\rightarrow_\beta (x[\![y/x]\!])[\![x/T]\!] \\ &\rightarrow x[\![x/T]\!] \\ &\rightarrow T\end{aligned}$$

Explicit substitutions

- In traditional λ-calculus the substitution is made all at once.

$$(\lambda x.(\lambda y.x)\ x)T \ \rightarrow_\beta \ ((\lambda y.x)\ x)[\![x/T]\!] = ((\lambda y.T)\ T)$$

- This can lead to a size explosion when having multiple copies of a variable.
- In practice, we need more control over what's going on during the substitution.

$$\begin{array}{lll} (\lambda x.(\lambda y.x)\ x)T & \rightarrow_\beta & ((\lambda y.x)\ x)[\![x/T]\!] \\ & \rightarrow_\beta & (x[\![y/x]\!])[\![x/T]\!] \\ & \rightarrow & x[\![x/T]\!] \\ & \rightarrow & T \end{array}$$

Presenting λx

Presented by K. Rose in 1993, λx is the shortest, simpler calculus with explicit substitutions:

$$
\begin{array}{lllll}
\text{(B)} & (\lambda x.T)\ U & \longrightarrow & T[x/U] & \\
 & x[x/U] & \longrightarrow & U & \\
 & y[x/U] & \longrightarrow & y & (x \neq y) \\
 & (T\ U)[x/V] & \longrightarrow & T[x/V]\ U[x/V] & \\
 & (\lambda y.T)[x/V] & \longrightarrow & \lambda y.T[x/V] &
\end{array}
$$

Note: The substitution is now part of the terms. We can construct the term $x[y/w]$ and start the computation from there.

Presenting λx

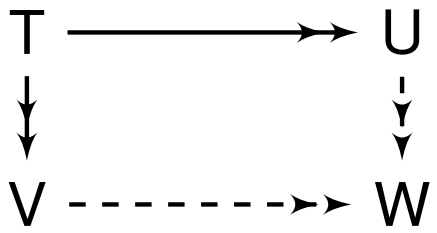
Presented by K. Rose in 1993, λx is the shortest, simpler calculus with explicit substitutions:

$$
\begin{array}{lllll}
\text{(B)} & (\lambda x.T)\ U & \longrightarrow & T[x/U] & \\
 & x[x/U] & \longrightarrow & U & \\
 & y[x/U] & \longrightarrow & y & (x \neq y) \\
 & (T\ U)[x/V] & \longrightarrow & T[x/V]\ U[x/V] & \\
 & (\lambda y.T)[x/V] & \longrightarrow & \lambda y.T[x/V] &
\end{array}
$$

Note: The substitution is now part of the terms. We can construct the term $x[y/w]$ and start the computation from there.

Properties of λx

- λx is *confluent*:

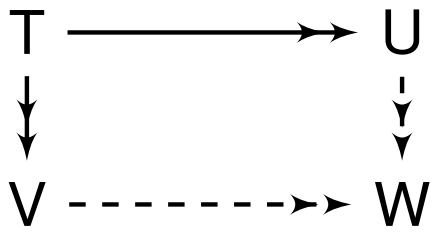


- A term is **strongly normalizing** if there is no infinite computation from it.
  λx *preserves strong normalization*:
  All the terms that are *strongly normalizing* in λ-calculus are also strongly normalizing in λx.

- λx is **not** *metaconfluent*: If we add *metavariables* (X, Y, ...) to the calculus, then confluence is lost.

Properties of λx

- λx is *confluent*:

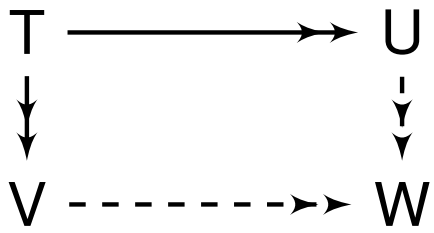


- A term is **strongly normalizing** if there is no infinite computation from it.
  λx *preserves strong normalization*:
  All the terms that are *strongly normalizing* in λ-calculus are also strongly normalizing in λx.

- λx is **not** *metaconfluent*: If we add *metavariables* ($X, Y, \ldots$) to the calculus, then confluence is lost.

Properties of λx

- λx is *confluent*:

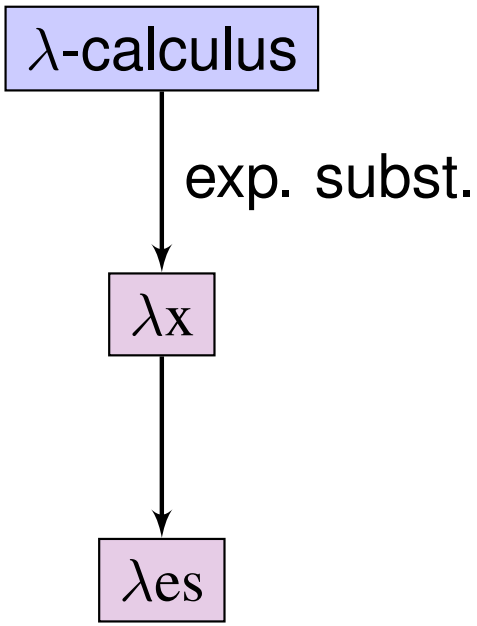


- A term is **strongly normalizing** if there is no infinite computation from it.
  λx *preserves strong normalization*:
  All the terms that are *strongly normalizing* in λ-calculus are also strongly normalizing in λx.

- λx is **not** *metaconfluent*: If we add *metavariables* (X, Y, ...) to the calculus, then confluence is lost.

Part of the λ-calculus hierarchy

What we are going to see



λes essentials

The calculus λes was created in 2007 by Delia Kesner. It gets preservation of strong normalization and metaconfluence by mixing two ideas:

- Composition of substitutions:
  When having two substitutions together, put the second in the first one:

$$T[x/U][y/V] \rightarrow T[y/V][x/U[y/V]]$$

  Needed to get metaconfluence. If no restriction is specified, the preservation of strong normalization is lost.

λes essentials

The calculus λes was created in 2007 by Delia Kesner. It gets preservation of strong normalization and metaconfluence by mixing two ideas:

- Composition of substitutions:
  When having two substitutions together, put the second in the first one:

$$T[x/U][y/V] \rightarrow T[y/V][x/U[y/V]]$$

  Needed to get metaconfluence. If no restriction is specified, the preservation of strong normalization is lost.

λes essentials

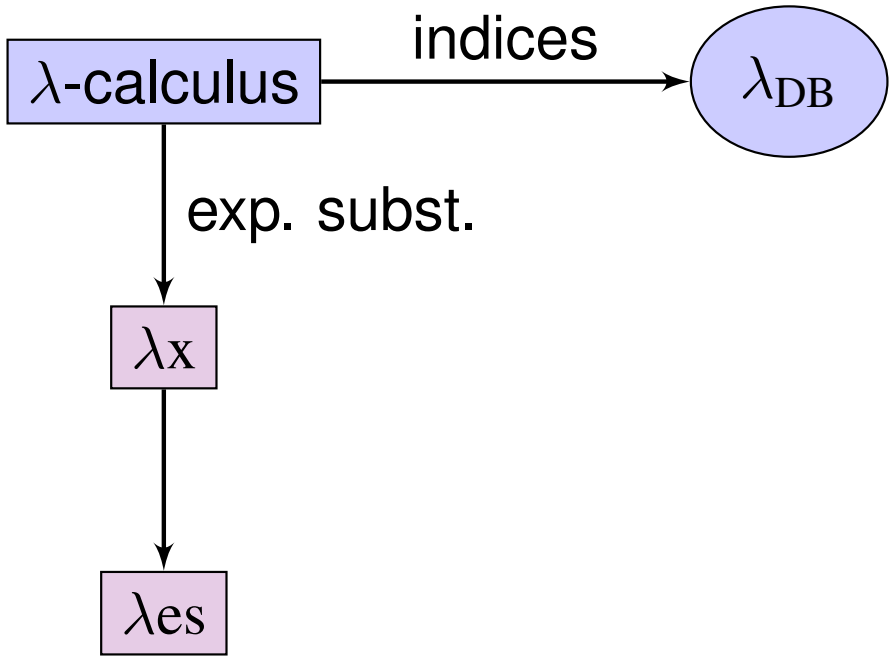
- Garbage collection:
  A substitution $[x/U]$ is *garbage* for the term T if x is not free in T.
  The idea is that T will remain unchanged after performing the substitution, *i. e.*

$$T[x/U] \rightarrow \ldots \rightarrow T$$

  Needed to control the composition, avoiding composition of garbage.

Part of the λ-calculus hierarchy

What we are going to see



Presenting λ_{DB}

Created by N. G. de Bruijn in 1972 for his Automath theorem prover.

- Motivation: knowing whether two terms are "the same" without making variable replacements. Remember the notion of α-congruence:

$$(\lambda x.(\lambda y.x\ y)) \equiv_\alpha (\lambda w.(\lambda z.w\ z))$$

- Idea: for each variable, count how many λ's there are in between, and replace the variable by this number.

$$(\lambda x.(\lambda y.x\ y\ v)) \quad \text{is translated to} \quad (\lambda(\lambda 2\ 1\ 3))$$

$$(\lambda w.(\lambda z.w\ z\ v)) \quad \text{is translated to} \quad (\lambda(\lambda 2\ 1\ 3))$$

Presenting λ_{DB}

Created by N. G. de Bruijn in 1972 for his Automath theorem prover.

- Motivation: knowing whether two terms are "the same" without making variable replacements. Remember the notion of α-congruence:

$$(\lambda x.(\lambda y.x\ y)) \equiv_\alpha (\lambda w.(\lambda z.w\ z))$$

- Idea: for each variable, count how many λ's there are in between, and replace the variable by this number.

$$(\lambda x.(\lambda y.x\ y\ v)) \quad \text{is translated to} \quad (\lambda(\lambda 2\ 1\ 3))$$

$$(\lambda w.(\lambda z.w\ z\ v)) \quad \text{is translated to} \quad (\lambda(\lambda 2\ 1\ 3))$$

Presenting λ_{DB}

Advantages:

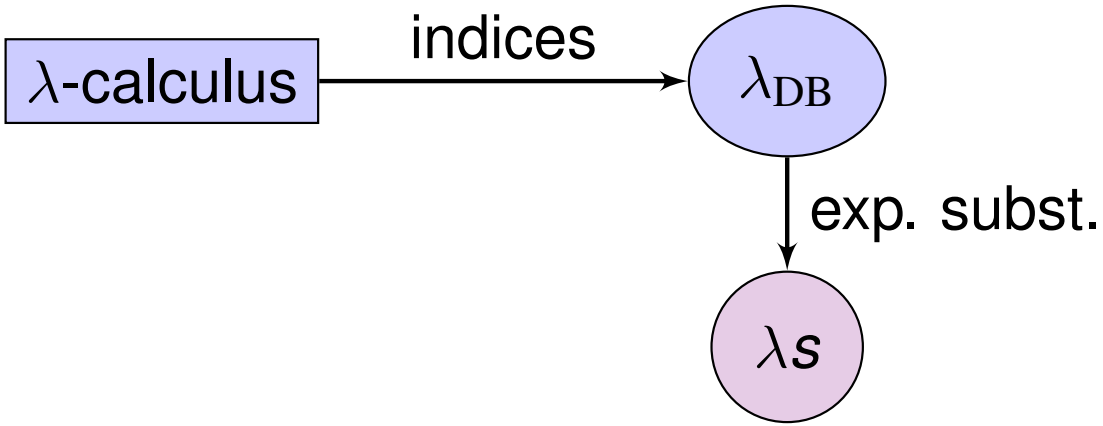
All the terms of the same class are encoded to the same λ_{DB}-term.

Disadvantages:

Indices need to be updated:

- Each time we perform a β-reduction.
- Each time we perform a substitution under a λ.

A word about λs

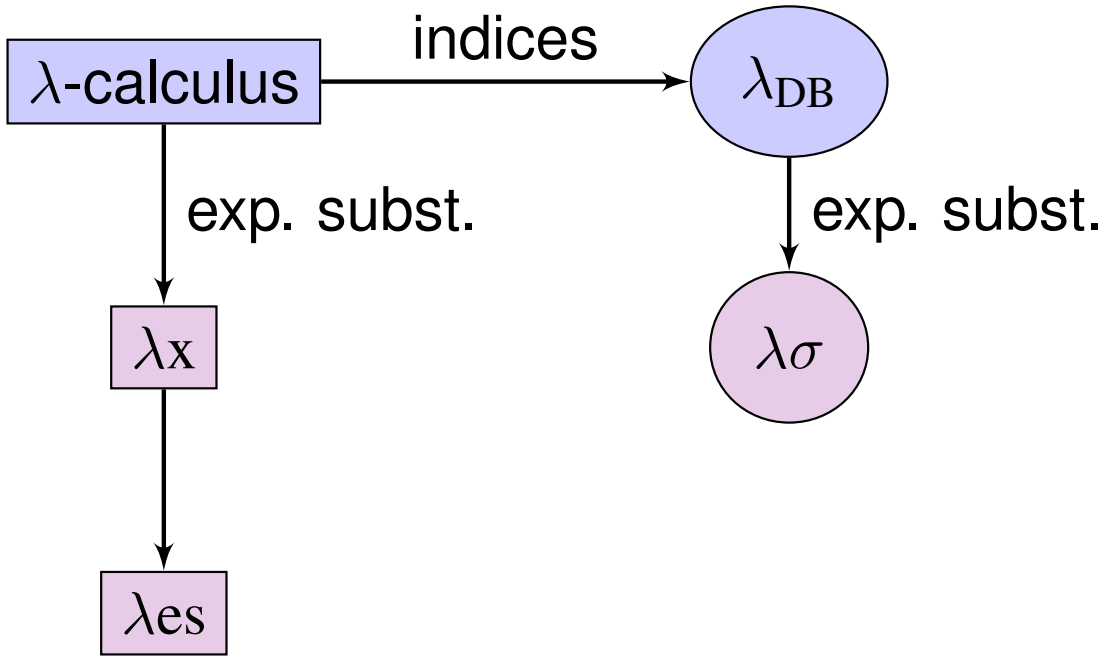


In 1995, F. Kamareddine and A. Ríos proposed a version of λ_{DB} with explicit substitutions, λs.

- As λx, is a direct translation of the rules into exp. sust.
- Holds the same properties as λx.
- Because of the indices updating mechanism, it is hard to extend it with composition of substitutions.

Part of the λ-calculus hierarchy

What we are going to see



Indices and composition of substitutions: $\lambda\sigma$

$\lambda\sigma$ was introduced by Abadi, Cardelli, Curien, and Levy in 1991.
The main features are:

- Multiple substitutions:
  Substitutions are a list $(1/a_1, 2/a_2, \ldots, n/a_n)$
- Born with composition of substitutions in mind.
- Indices updating is made in an atomic way.

But:

- It does not preserve strong normalization.
- It is not metaconfluent (but there are extensions that are, or are semi-metaconfluent).

Indices and composition of substitutions: $\lambda\sigma$

$\lambda\sigma$ was introduced by Abadi, Cardelli, Curien, and Levy in 1991.
The main features are:

- Multiple substitutions:
  Substitutions are a list $(1/a_1, 2/a_2, \ldots, n/a_n)$
- Born with composition of substitutions in mind.
- Indices updating is made in an atomic way.

But:

- It does not preserve strong normalization.
- It is not metaconfluent (but there are extensions that are, or are semi-metaconfluent).

Indices and composition of substitutions: $\lambda\sigma$

$\lambda\sigma$ was introduced by Abadi, Cardelli, Curien, and Levy in 1991.
The main features are:

- Multiple substitutions:
  Substitutions are a list $(1/a_1, 2/a_2, \ldots, n/a_n)$
- Born with composition of substitutions in mind.
- Indices updating is made in an atomic way.

But:

- It does not preserve strong normalization.
- It is not metaconfluent (but there are extensions that are, or are semi-metaconfluent).

Indices and composition of substitutions: $\lambda\sigma$

$\lambda\sigma$ was introduced by Abadi, Cardelli, Curien, and Levy in 1991.
The main features are:

- Multiple substitutions:
  Substitutions are a list $(1/a_1, 2/a_2, \ldots, n/a_n)$
- Born with composition of substitutions in mind.
- Indices updating is made in an atomic way.

But:

- It does not preserve strong normalization.
- It is not metaconfluent (but there are extensions that are, or are semi-metaconfluent).

Indices and composition of substitutions: $\lambda\sigma$

$\lambda\sigma$ was introduced by Abadi, Cardelli, Curien, and Levy in 1991.
The main features are:

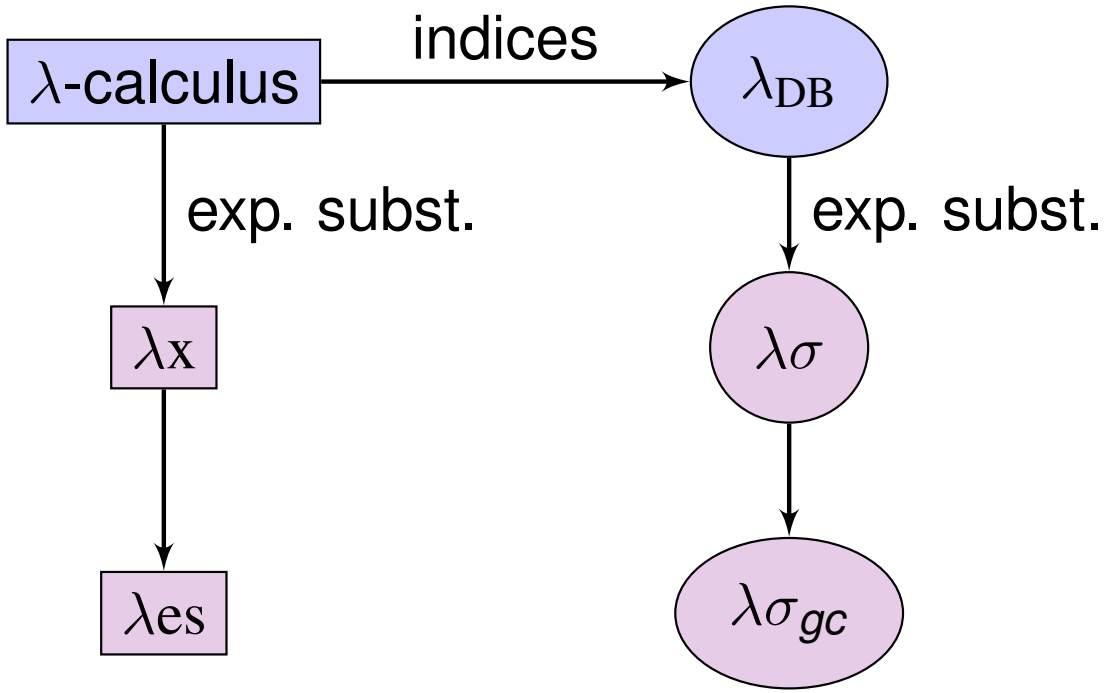
- Multiple substitutions:
  Substitutions are a list $(1/a_1, 2/a_2, \ldots, n/a_n)$
- Born with composition of substitutions in mind.
- Indices updating is made in an atomic way.

But:

- It does not preserve strong normalization.
- It is not metaconfluent (but there are extensions that are, or are semi-metaconfluent).

Part of the λ-calculus hierarchy

What we are going to see



In the search for PSN: $\lambda\sigma_{gc}$

Remember the definition of garbage in λes:
"The substitution is garbage if the variable to substitute does not appear free in the term"

This is difficult to specify in $\lambda\sigma$ for two reasons:

- Indices updating (applicable to all calculus with indices):
  The substitution is in charge of the substitution itself **and** the process of index updating, so it is not easy to remove.
- Multiple substitutions:
  All variables must be taken into account.

In the search for PSN: $\lambda\sigma_{gc}$

Remember the definition of garbage in λes:
"The substitution is garbage if the variable to substitute does not appear free in the term"

This is difficult to specify in $\lambda\sigma$ for two reasons:

- Indices updating (applicable to all calculus with indices):
  The substitution is in charge of the substitution itself **and** the process of index updating, so it is not easy to remove.
- Multiple substitutions:
  All variables must be taken into account.

In the search for PSN: $\lambda\sigma_{gc}$

Remember the definition of garbage in λes:
"The substitution is garbage if the variable to substitute does not appear free in the term"

This is difficult to specify in $\lambda\sigma$ for two reasons:

- Indices updating (applicable to all calculus with indices):
  The substitution is in charge of the substitution itself **and** the process of index updating, so it is not easy to remove.
- Multiple substitutions:
  All variables must be taken into account.

$\lambda\sigma_{gc}$: Results and open problems

What we do and what we know:

- First definition of garbage for $\lambda\sigma$.
- At least, the same good properties as $\lambda\sigma$.
- A known term that is a counterexample for no preservation of strong normalization in $\lambda\sigma$ is strongly normalizing in $\lambda\sigma_{gc}$.
- Is not metaconfluent.
- The definition of garbage is too restrictive.

What we don't know:

- Does it preserve strong normalization?
- Is there a trivial extension to obtain metaconfluence?
- Is there a better definition of garbage collection?

$\lambda\sigma_{gc}$: Results and open problems

What we do and what we know:

- First definition of garbage for $\lambda\sigma$.
- At least, the same good properties as $\lambda\sigma$.
- A known term that is a counterexample for no preservation of strong normalization in $\lambda\sigma$ is strongly normalizing in $\lambda\sigma_{gc}$.
- Is not metaconfluent.
- The definition of garbage is too restrictive.

What we don't know:

- Does it preserve strong normalization?
- Is there a trivial extension to obtain metaconfluence?
- Is there a better definition of garbage collection?

Summary

- There are a lot of calculi related to λ-calculus . Some of them use indices, others explicit substitutions. Some of them use both concepts.
- It is not easy to get preservation of strong normalization and metaconfluence.
- Garbage collection is needed, but for $\lambda\sigma$ it is hard to add .

The end

Questions?

Understunding λ_{DB} by example

$$(\lambda(\lambda 1\ 2\ 3))\ b \rightarrow_\beta (\lambda 1\ 2\ 3)[\![1/b]\!]$$

- $(\lambda a)\ b \rightarrow_\beta a[\![1/b]\!]$

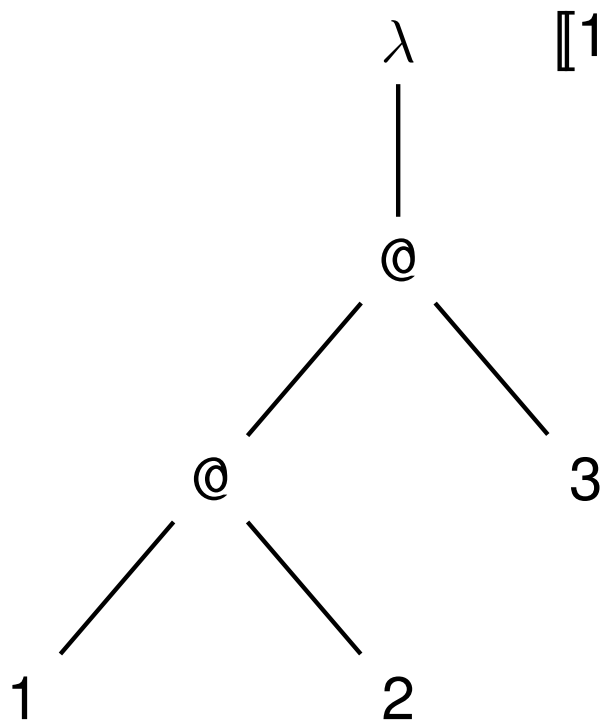
Understunding λ_{DB} by example

$$(\lambda(\lambda 1\ 2\ 3))\ b \ \rightarrow_\beta \ (\lambda 1\ 2\ 3)[\![1/b]\!]$$

- $(\lambda a)\ b \ \rightarrow_\beta \ a[\![1/b]\!]$

Understunding λ_{DB} by example

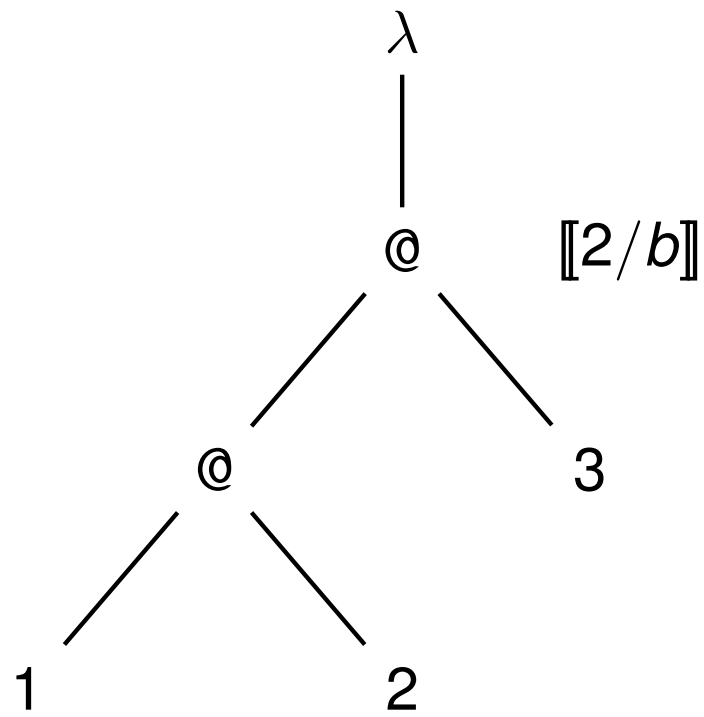
$$(\lambda(\lambda 1\ 2\ 3))\ b \ \rightarrow_\beta \ (\lambda 1\ 2\ 3)[\![1/b]\!]$$



- $(\lambda a)\ b \ \rightarrow_\beta \ a[\![1/b]\!]$

Understunding λ_{DB} by example

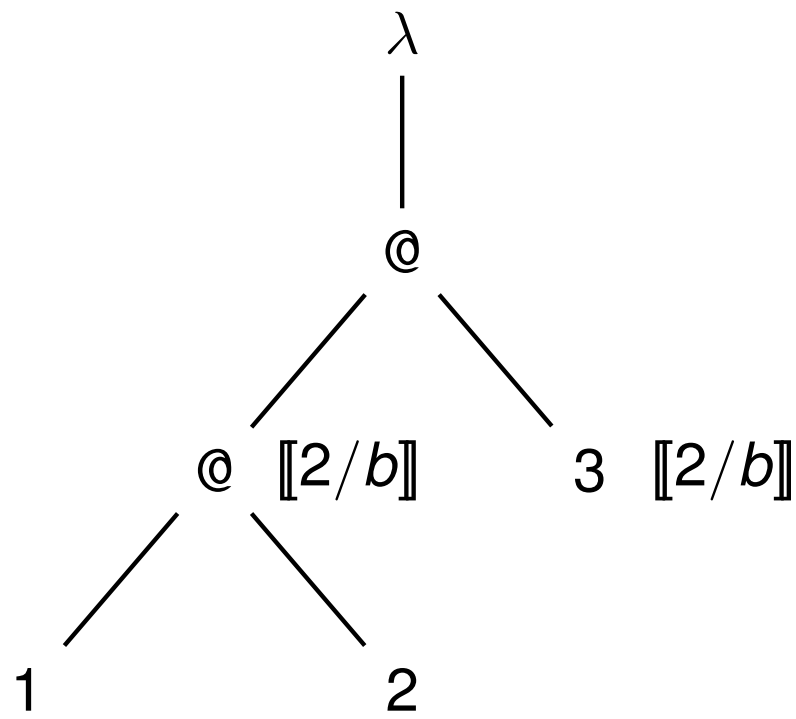
$$(\lambda(\lambda 1\ 2\ 3))\ b \ \rightarrow_\beta \ \lambda(1\ 2\ 3)[\![2/b]\!]$$



- $(\lambda a)\ b \ \rightarrow_\beta \ a[\![1/b]\!]$
- $(\lambda a)[\![i/b]\!] = \lambda a[\![i+1/b]\!]$

Understunding λ_{DB} by example

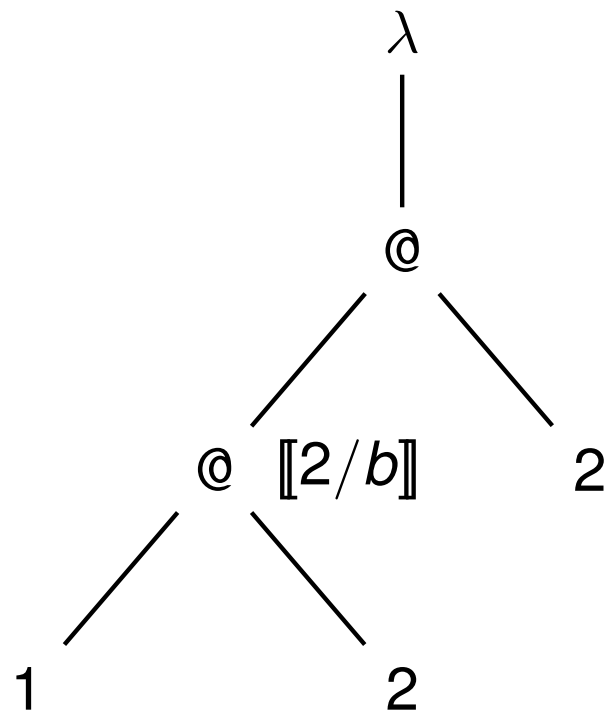
$$(\lambda(\lambda 1\ 2\ 3))\ b\ \rightarrow_\beta\ \lambda((1\ 2)[\![2/b]\!]\ 3[\![2/b]\!])$$



- $(\lambda a)\ b\ \rightarrow_\beta\ a[\![1/b]\!]$
- $(\lambda a)[\![i/b]\!] = \lambda a[\![i+1/b]\!]$
- $(a_1\ a_2)[\![i/b]\!] = a_1[\![i/b]\!]\ a_2[\![i/b]\!]$

Understunding λ_{DB} by example

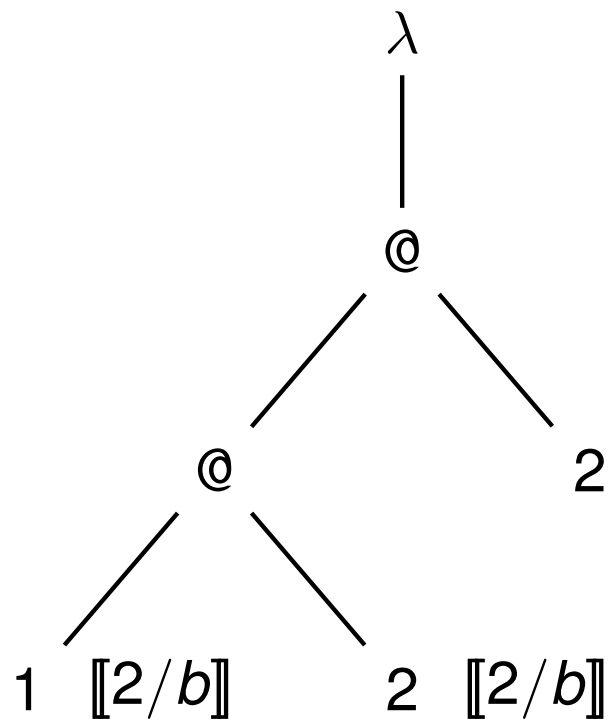
$$(\lambda(\lambda 1\ 2\ 3))\ b\ \rightarrow_\beta\ \lambda((1\ 2)[\![2/b]\!]\ 2)$$



- $(\lambda a)\ b\ \rightarrow_\beta\ a[\![1/b]\!]$
- $(\lambda a)[\![i/b]\!] = \lambda a[\![i+1/b]\!]$
- $(a_1\ a_2)[\![i/b]\!] = a_1[\![i/b]\!]\ a_2[\![i/b]\!]$
- $j[\![i/b]\!] = j - i \quad (j > i)$

Understunding λ_{DB} by example

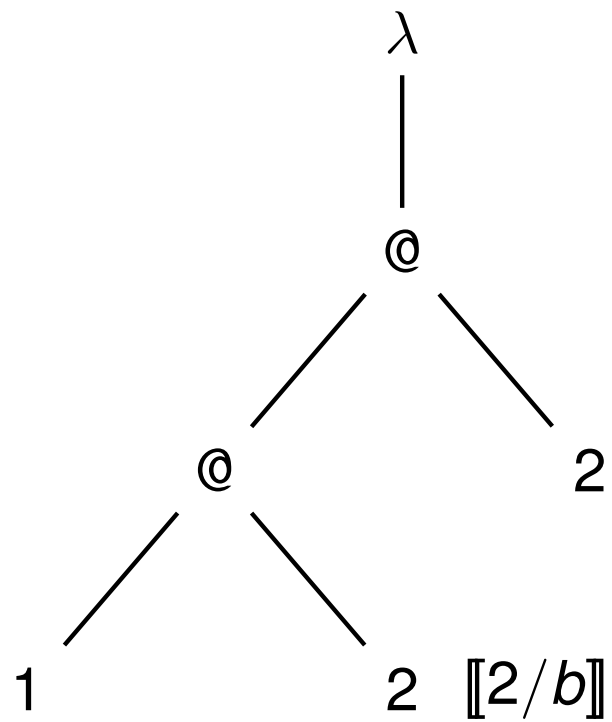
$$(\lambda(\lambda 1\ 2\ 3))\ b\ \rightarrow_\beta\ \lambda(1[\![2/b]\!]\ 2[\![2/b]\!]\ 2)$$



- $(\lambda a)\ b\ \rightarrow_\beta\ a[\![1/b]\!]$
- $(\lambda a)[\![i/b]\!] = \lambda a[\![i+1/b]\!]$
- $(a_1\ a_2)[\![i/b]\!] = a_1[\![i/b]\!]\ a_2[\![i/b]\!]$
- $j[\![i/b]\!] = j - i \quad (j > i)$

Understunding λ_{DB} by example

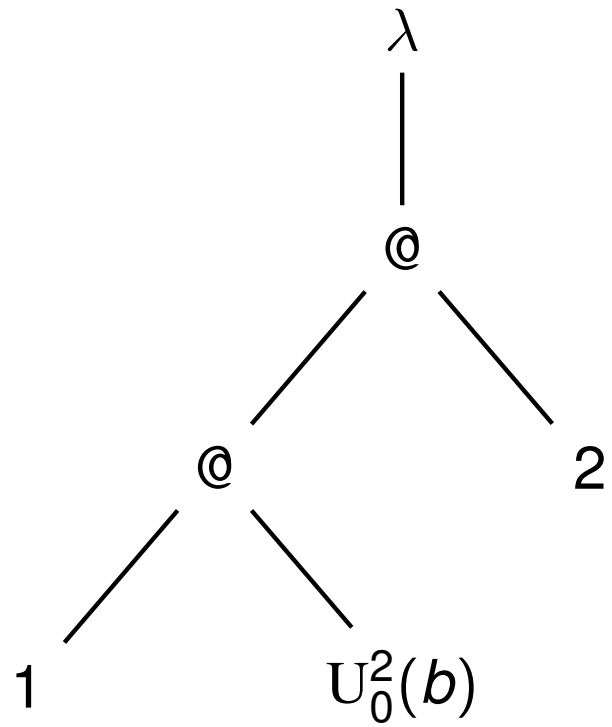
$$(\lambda(\lambda 1\ 2\ 3))\ b \ \rightarrow_\beta \ \lambda(1\ 2[\![2/b]\!]\ 2)$$



- $(\lambda a)\ b \ \rightarrow_\beta \ a[\![1/b]\!]$
- $(\lambda a)[\![i/b]\!] = \lambda a[\![i+1/b]\!]$
- $(a_1\ a_2)[\![i/b]\!] = a_1[\![i/b]\!]\ a_2[\![i/b]\!]$
- $j[\![i/b]\!] = j - i \quad (j > i)$
- $j[\![i/b]\!] = j \quad (j < i)$

Understunding λ_{DB} by example

$$(\lambda(\lambda 1\ 2\ 3))\ b\ \rightarrow_\beta\ \lambda(1\ \mathrm{U}_0^2(b)\ 2)$$



- $(\lambda a)\ b\ \rightarrow_\beta\ a[\![1/b]\!]$
- $(\lambda a)[\![i/b]\!] = \lambda a[\![i+1/b]\!]$
- $(a_1\ a_2)[\![i/b]\!] = a_1[\![i/b]\!]\ a_2[\![i/b]\!]$
- $j[\![i/b]\!] = j - i \quad (j > i)$
- $j[\![i/b]\!] = j \quad (j < i)$
- $j[\![i/b]\!] = \mathrm{U}_0^i(b) \quad (j = i)$

Understunding λ_{DB}

The update operator

The idea of the update operator $\mathrm{U}_k^i(b)$ is:

- Leave al bounded variables of b untouched.
- Update the free variables by adding $i-1$, the number of λ's crossed.

So

- The parameter i never changes.
- The parameter k is increased each time we cross a λ.