



Higher Order Abstract Syntax and λ -Calculus

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

Abstract Syntax for Variables

Recall the `let` construct we added in the last lecture. We shall extend our AST and parsing relation to include a definition for `let` and variables.

Let Syntax

$$\begin{array}{c}
 x \text{ Ident} \\
 \hline
 x \text{ Atom} \longleftrightarrow (\text{Var } x) \text{ AST} \\
 \\
 \frac{x \text{ Ident} \quad e_1 \text{ SExp} \longleftrightarrow a_1 \text{ AST} \quad e_2 \text{ SExp} \longleftrightarrow a_2 \text{ AST}}{\text{let } x = e_1 \text{ in } e_2 \text{ end Atom} \longleftrightarrow (\text{Let } x \ a_1 \ a_2) \text{ AST}}
 \end{array}$$

First Order Abstract Syntax

Consider the following two pieces of abstract syntax:

```
(Let "x" (Num 5) (Plus (Num 4) (Var "x")))
```

```
(Let "y" (Num 5) (Plus (Num 4) (Var "y")))
```

This demonstrates some problems with our abstract syntax approach.

- 1 Substitution capture is a problem.
- 2 α -equivalent expressions are **not equal**. Determining if an expression is α -equivalent requires us to search for a consistent α -renaming of variables.
- 3 No distinction is made between **binding** and **usage** occurrences of variables. This means that we must define substitution by hand on each type of expression we introduce.
- 4 Scoping errors cannot be easily detected — **malformed syntax** is easy to write.

First Order Abstract Syntax

Consider the following two pieces of abstract syntax:

```
(Let "x" (Num 5) (Plus (Num 4) (Var "x")))
```

```
(Let "y" (Num 5) (Plus (Num 4) (Var "y")))
```

This demonstrates some problems with our abstract syntax approach.

- 1 Substitution capture is a problem.
- 2 α -equivalent expressions are **not equal**. Determining if an expression is α -equivalent requires us to search for a consistent α -renaming of variables.
- 3 No distinction is made between **binding** and **usage** occurrences of variables. This means that we must define substitution by hand on each type of expression we introduce.
- 4 Scoping errors cannot be easily detected — **malformed syntax** is easy to write.

First Order Abstract Syntax

Consider the following two pieces of abstract syntax:

```
(Let "x" (Num 5) (Plus (Num 4) (Var "x")))
```

```
(Let "y" (Num 5) (Plus (Num 4) (Var "y")))
```

This demonstrates some problems with our abstract syntax approach.

- 1 Substitution capture is a problem.
- 2 α -equivalent expressions are **not equal**. Determining if an expression is α -equivalent requires us to search for a consistent α -renaming of variables.
- 3 No distinction is made between **binding** and **usage** occurrences of variables. This means that we must define substitution by hand on each type of expression we introduce.
- 4 Scoping errors cannot be easily detected — **malformed syntax** is easy to write.

First Order Abstract Syntax

Consider the following two pieces of abstract syntax:

```
(Let "x" (Num 5) (Plus (Num 4) (Var "x")))
```

```
(Let "y" (Num 5) (Plus (Num 4) (Var "y")))
```

This demonstrates some problems with our abstract syntax approach.

- ① Substitution capture is a problem.
- ② α -equivalent expressions are **not equal**. Determining if an expression is α -equivalent requires us to search for a consistent α -renaming of variables.
- ③ No distinction is made between **binding** and **usage** occurrences of variables. This means that we must define substitution by hand on each type of expression we introduce.
- ④ Scoping errors cannot be easily detected — **malformed syntax** is easy to write.

First Order Abstract Syntax

Consider the following two pieces of abstract syntax:

```
(Let "x" (Num 5) (Plus (Num 4) (Var "x")))
```

```
(Let "y" (Num 5) (Plus (Num 4) (Var "y")))
```

This demonstrates some problems with our abstract syntax approach.

- ① Substitution capture is a problem.
- ② α -equivalent expressions are **not equal**. Determining if an expression is α -equivalent requires us to search for a consistent α -renaming of variables.
- ③ No distinction is made between **binding** and **usage** occurrences of variables. This means that we must define substitution by hand on each type of expression we introduce.
- ④ Scoping errors cannot be easily detected — **malformed syntax** is easy to write.

de Bruijn Indices

One popular approach to address the first issue is *de Bruijn indices*.

Key Idea

- 1 Remove all identifiers from binding expressions like Let.
- 2 Replace the identifier in a Var with a number indicating how many binders we must skip in order to find the binder for that variable.

```
(Let "a" (Num 5)
  (Let "y" (Num 2)
    (Plus (Var "a") (Var "y"))))
```

```
(Let (Num 5)
  (Let (Num 2)
    (Plus (Var 1) (Var 0))))
```


de Bruijn Indices

One popular approach to address the first issue is *de Bruijn indices*.

Key Idea

- 1 Remove all identifiers from binding expressions like Let.
- 2 Replace the identifier in a Var with a number indicating how many binders we must skip in order to find the binder for that variable.

<pre>(Let "a" (Num 5) (Let "y" (Num 2) (Plus (Var "a") (Var "y"))))</pre>	<pre>(Let (Num 5) (Let (Num 2) (Plus (Var 1) (Var 0))))</pre>
--	--

de Bruijn Indices

One popular approach to address the first issue is *de Bruijn indices*.

Key Idea

- 1 Remove all identifiers from binding expressions like Let.
- 2 Replace the identifier in a Var with a number indicating how many binders we must skip in order to find the binder for that variable.

```
(Let "a" (Num 5)
 (Let "y" (Num 2)
  (Plus (Var "a") (Var "y"))))
```

```
(Let (Num 5)
 (Let (Num 2)
  (Plus (Var 1) (Var 0))))
```

Debruijnification

Algorithm

Given a piece of *first order abstract syntax* with explicit variable names, we can convert to de Bruijn indices by keeping a *stack* of variable names, pushing onto the stack at each Let and popping after the variable goes out of scope. When a usage occurrence is encountered, replace the variable name with its *first position* in the stack (starting at the top of the stack).

This approach naturally handles *shadowing*. It's also possible, but harder, to have de Bruijn indices going in the other direction (from the bottom of the stack, upwards).

de Bruijn Substitution

Substitution is now **capture avoiding** by definition.

$$\begin{aligned}
 (\text{Num } i)[n := t] &= (\text{Num } i) \\
 (\text{Plus } a \ b)[n := t] &= (\text{Plus } a[n := t] \ b[n := t]) \\
 (\text{Times } a \ b)[n := t] &= (\text{Times } a[n := t] \ b[n := t]) \\
 (\text{Var } m)[n := t] &= \begin{cases} t & \text{if } n = m \\ (\text{Var } (m - 1)) & \text{if } m > n \\ (\text{Var } m) & \text{otherwise} \end{cases} \\
 (\text{Let } e_1 \ e_2)[n := t] &= (\text{Let } e_1[n := t] \ e_2[n + 1 := t_{\uparrow 0}])
 \end{aligned}$$

Where $e_{\uparrow n}$ is an **up-shifting** operation defined as follows:

$$\begin{aligned}
 (\text{Num } i)_{\uparrow n} &= (\text{Num } i) \\
 (\text{Plus } a \ b)_{\uparrow n} &= (\text{Plus } a_{\uparrow n} \ b_{\uparrow n}) \\
 (\text{Times } a \ b)_{\uparrow n} &= (\text{Times } a_{\uparrow n} \ b_{\uparrow n}) \\
 (\text{Var } m)_{\uparrow n} &= \begin{cases} (\text{Var } (m + 1)) & \text{if } m \geq n \\ (\text{Var } m) & \text{otherwise} \end{cases} \\
 (\text{Let } e_1 \ e_2)_{\uparrow n} &= (\text{Let } e_{1\uparrow n} \ e_{2\uparrow n+1})
 \end{aligned}$$

de Bruijn Substitution

Substitution is now **capture avoiding** by definition.

$$\begin{aligned}
 (\text{Num } i)[n := t] &= (\text{Num } i) \\
 (\text{Plus } a \ b)[n := t] &= (\text{Plus } a[n := t] \ b[n := t]) \\
 (\text{Times } a \ b)[n := t] &= (\text{Times } a[n := t] \ b[n := t]) \\
 (\text{Var } m)[n := t] &= \begin{cases} t & \text{if } n = m \\ (\text{Var } (m - 1)) & \text{if } m > n \\ (\text{Var } m) & \text{otherwise} \end{cases} \\
 (\text{Let } e_1 \ e_2)[n := t] &= (\text{Let } e_1[n := t] \ e_2[n + 1 := t_{\uparrow 0}])
 \end{aligned}$$

Where $e_{\uparrow n}$ is an *up-shifting* operation defined as follows:

$$\begin{aligned}
 (\text{Num } i)_{\uparrow n} &= (\text{Num } i) \\
 (\text{Plus } a \ b)_{\uparrow n} &= (\text{Plus } a_{\uparrow n} \ b_{\uparrow n}) \\
 (\text{Times } a \ b)_{\uparrow n} &= (\text{Times } a_{\uparrow n} \ b_{\uparrow n}) \\
 (\text{Var } m)_{\uparrow n} &= \begin{cases} (\text{Var } (m + 1)) & \text{if } m \geq n \\ (\text{Var } m) & \text{otherwise} \end{cases} \\
 (\text{Let } e_1 \ e_2)_{\uparrow n} &= (\text{Let } e_{1\uparrow n} \ e_{2\uparrow n+1})
 \end{aligned}$$

de Bruijn Substitution

Substitution is now **capture avoiding** by definition.

$$\begin{aligned}
 (\text{Num } i)[n := t] &= (\text{Num } i) \\
 (\text{Plus } a \ b)[n := t] &= (\text{Plus } a[n := t] \ b[n := t]) \\
 (\text{Times } a \ b)[n := t] &= (\text{Times } a[n := t] \ b[n := t]) \\
 (\text{Var } m)[n := t] &= \begin{cases} t & \text{if } n = m \\ (\text{Var } (m - 1)) & \text{if } m > n \\ (\text{Var } m) & \text{otherwise} \end{cases} \\
 (\text{Let } e_1 \ e_2)[n := t] &= (\text{Let } e_1[n := t] \ e_2[n + 1 := t_{\uparrow 0}])
 \end{aligned}$$

Where $e_{\uparrow n}$ is an **up-shifting** operation defined as follows:

$$\begin{aligned}
 (\text{Num } i)_{\uparrow n} &= (\text{Num } i) \\
 (\text{Plus } a \ b)_{\uparrow n} &= (\text{Plus } a_{\uparrow n} \ b_{\uparrow n}) \\
 (\text{Times } a \ b)_{\uparrow n} &= (\text{Times } a_{\uparrow n} \ b_{\uparrow n}) \\
 (\text{Var } m)_{\uparrow n} &= \begin{cases} (\text{Var } (m + 1)) & \text{if } m \geq n \\ (\text{Var } m) & \text{otherwise} \end{cases} \\
 (\text{Let } e_1 \ e_2)_{\uparrow n} &= (\text{Let } e_{1\uparrow n} \ e_{2\uparrow n+1})
 \end{aligned}$$

de Bruijn Substitution

Substitution is now **capture avoiding** by definition.

$$\begin{aligned}
 (\text{Num } i)[n := t] &= (\text{Num } i) \\
 (\text{Plus } a \ b)[n := t] &= (\text{Plus } a[n := t] \ b[n := t]) \\
 (\text{Times } a \ b)[n := t] &= (\text{Times } a[n := t] \ b[n := t]) \\
 (\text{Var } m)[n := t] &= \begin{cases} t & \text{if } n = m \\ (\text{Var } (m - 1)) & \text{if } m > n \\ (\text{Var } m) & \text{otherwise} \end{cases} \\
 (\text{Let } e_1 \ e_2)[n := t] &= (\text{Let } e_1[n := t] \ e_2[n + 1 := t_{\uparrow 0}])
 \end{aligned}$$

Where $e_{\uparrow n}$ is an **up-shifting** operation defined as follows:

$$\begin{aligned}
 (\text{Num } i)_{\uparrow n} &= (\text{Num } i) \\
 (\text{Plus } a \ b)_{\uparrow n} &= (\text{Plus } a_{\uparrow n} \ b_{\uparrow n}) \\
 (\text{Times } a \ b)_{\uparrow n} &= (\text{Times } a_{\uparrow n} \ b_{\uparrow n}) \\
 (\text{Var } m)_{\uparrow n} &= \begin{cases} (\text{Var } (m + 1)) & \text{if } m \geq n \\ (\text{Var } m) & \text{otherwise} \end{cases} \\
 (\text{Let } e_1 \ e_2)_{\uparrow n} &= (\text{Let } e_{1\uparrow n} \ e_{2\uparrow n+1})
 \end{aligned}$$

Examining de Bruijn indices

How do de Bruijn indices stack up against our explicit names in terms of the problems we identified?

- ① Substitution capture **solved**.
- ② α -equivalent expressions are now **equal**.
- ③ We still must define substitution machinery **by hand** for each type of expression.
- ④ It is still possible to make **malformed syntax** – indices that overflow the stack, for example.

Two out of four isn't bad, but can we do better by changing the **term language**?

Examining de Bruijn indices

How do de Bruijn indices stack up against our explicit names in terms of the problems we identified?

- ① Substitution capture **solved**.
- ② α -equivalent expressions are now **equal**.
- ③ We still must define substitution machinery **by hand** for each type of expression.
- ④ It is still possible to make **malformed syntax** – indices that overflow the stack, for example.

Two out of four isn't bad, but can we do better by changing the **term language**?

Examining de Bruijn indices

How do de Bruijn indices stack up against our explicit names in terms of the problems we identified?

- ① Substitution capture **solved**.
- ② α -equivalent expressions are now **equal**.
- ③ We still must define substitution machinery **by hand** for each type of expression.
- ④ It is still possible to make **malformed syntax** – indices that overflow the stack, for example.

Two out of four isn't bad, but can we do better by changing the **term language**?

Examining de Bruijn indices

How do de Bruijn indices stack up against our explicit names in terms of the problems we identified?

- ① Substitution capture **solved**.
- ② α -equivalent expressions are now **equal**.
- ③ We still must define substitution machinery **by hand** for each type of expression.
- ④ It is still possible to make **malformed syntax** – indices that overflow the stack, for example.

Two out of four isn't bad, but can we do better by changing the **term language**?

Examining de Bruijn indices

How do de Bruijn indices stack up against our explicit names in terms of the problems we identified?

- ① Substitution capture **solved**.
- ② α -equivalent expressions are now **equal**.
- ③ We still must define substitution machinery **by hand** for each type of expression.
- ④ It is still possible to make **malformed syntax** – indices that overflow the stack, for example.

Two out of four isn't bad, but can we do better by changing the **term language**?

Higher Order Terms

We shall change our term language to include **built-in** notions of variables and binding.

$t ::=$	Symbol	(symbols)
	x	(variables)
	$t_1 t_2$	(application)
	$x. t$	(binding or abstraction)

As in Haskell, we shall say that application is **left-associative**, so

$$(\text{Plus} (\text{Num } 3) (\text{Num } 4)) = ((\text{Plus} (\text{Num } 3)) (\text{Num } 4))$$

Now the **binding** and **usage** occurrences of variables are distinguished from regular symbols in our term language. Let's see what this lets us do...

Representing Let

$$\frac{a_1 \text{ AST} \quad a_2 \text{ AST}}{(\text{Let } a_1 (x. a_2)) \text{ AST}}$$

We no longer need a rule for variables, because they're baked into the structure of terms.

How would we represent this AST in Haskell?

```
data AST = Num Int
         | Plus AST AST
         | Times AST AST
         | Let AST
```

So `let x = 3 in x + 2 end` becomes, in Haskell:

```
(Let (Num 3) (λx → Plus x (Num 2)))
```

Representing Let

$$\frac{a_1 \text{ AST} \quad a_2 \text{ AST}}{(\text{Let } a_1 (x. a_2)) \text{ AST}}$$

We no longer need a rule for variables, because they're baked into the structure of terms.

How would we represent this AST in **Haskell**?

```
data AST = Num Int
          | Plus AST AST
          | Times AST AST
          | Let AST ???
```

So `let x = 3 in x + 2 end` becomes, in Haskell:

```
(Let (Num 3) (λx → Plus x (Num 2)))
```

Representing Let

$$\frac{a_1 \text{ AST} \quad a_2 \text{ AST}}{(\text{Let } a_1 (x. a_2)) \text{ AST}}$$

We no longer need a rule for variables, because they're baked into the structure of terms.

How would we represent this AST in **Haskell**?

```
data AST = Num Int
          | Plus AST AST
          | Times AST AST
          | Let AST (AST → AST)
```

So `let x = 3 in x + 2 end` becomes, in Haskell:

```
(Let (Num 3) (λx → Plus x (Num 2)))
```


Substitution

We can now define substitution across **all** terms **in the meta-logic**:

$$\begin{aligned}
 \text{Symbol}[x := e] &= \text{Symbol} \\
 y[x := e] &= \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
 (t_1 \ t_2)[x := e] &= t_1[x := e] \ t_2[x := e] \\
 (y. \ t)[x := e] &= \begin{cases} (y. \ t) & \text{if } x = y \\ (y. \ t[x := e]) & \text{if } y \notin \text{FV}(e) \\ \text{undefined} & \text{otherwise} \end{cases}
 \end{aligned}$$

Where $\text{FV}(\cdot)$ is the set of all **free variables** in a term:

$$\begin{aligned}
 \text{FV}(\text{Symbol}) &= \emptyset \\
 \text{FV}(x) &= \{x\} \\
 \text{FV}(t_1 \ t_2) &= \text{FV}(t_1) \cup \text{FV}(t_2) \\
 \text{FV}(x. \ t) &= \text{FV}(t) \setminus \{x\}
 \end{aligned}$$

Substitution

We can now define substitution across **all** terms **in the meta-logic**:

$$\begin{aligned}
 \text{Symbol}[x := e] &= \text{Symbol} \\
 y[x := e] &= \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
 (t_1 \ t_2)[x := e] &= t_1[x := e] \ t_2[x := e] \\
 (y. \ t)[x := e] &= \begin{cases} (y. \ t) & \text{if } x = y \\ (y. \ t[x := e]) & \text{if } y \notin \text{FV}(e) \\ \text{undefined} & \text{otherwise} \end{cases}
 \end{aligned}$$

Where $\text{FV}(\cdot)$ is the set of all **free variables** in a term:

$$\begin{aligned}
 \text{FV}(\text{Symbol}) &= \emptyset \\
 \text{FV}(x) &= \{x\} \\
 \text{FV}(t_1 \ t_2) &= \text{FV}(t_1) \cup \text{FV}(t_2) \\
 \text{FV}(x. \ t) &= \text{FV}(t) \setminus \{x\}
 \end{aligned}$$

Substitution

We can now define substitution across **all** terms **in the meta-logic**:

$$\begin{aligned}
 \text{Symbol}[x := e] &= \text{Symbol} \\
 y[x := e] &= \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
 (t_1 \ t_2)[x := e] &= t_1[x := e] \ t_2[x := e] \\
 (y. \ t)[x := e] &= \begin{cases} (y. \ t) & \text{if } x = y \\ (y. \ t[x := e]) & \text{if } y \notin \text{FV}(e) \\ \text{undefined} & \text{otherwise} \end{cases}
 \end{aligned}$$

Where $\text{FV}(\cdot)$ is the set of all **free variables** in a term:

$$\begin{aligned}
 \text{FV}(\text{Symbol}) &= \emptyset \\
 \text{FV}(x) &= \{x\} \\
 \text{FV}(t_1 \ t_2) &= \text{FV}(t_1) \cup \text{FV}(t_2) \\
 \text{FV}(x. \ t) &= \text{FV}(t) \setminus \{x\}
 \end{aligned}$$

Cheating Outrageously

Substitution **capture** is still a problem in the substitution we just defined but it is **not our problem**. Because substitution is defined in the **meta-language**, it's the job of the implementors of the meta-language (if any) to deal with issues about capture.

- When **Haskell** is our meta-language, it's the job of the GHC developers to sort out capture.
- When we are doing proofs in our **meta-logic**, there is no implementation, so we can just say that we assume α -equivalent terms to be equal, and therefore assume that variables are always renamed to avoid capture.

So, we have solved the problem by making it someone else's problem. **Outrageous cheating!**

Evaluating All Approaches

	HOAS		FOAS	
	Proofs	Haskell	Strings	de Bruijn
Capture	Cheat	Cheat	Problem	Solved
α -equivalence	Cheat	Cheat	Problem	Solved
Generic subst.	Solved	Solved	Problem	Problem
Malformed syntax	Cheat	Cheat	Problem	Problem

- In **embedded languages** and in **proofs**, HOAS is very common.
- In conventional language implementations and machine-checked formalisations, de Bruijn indices are more popular.
- In your assignments, strings will be used

Evaluating All Approaches

	HOAS		FOAS	
	Proofs	Haskell	Strings	de Bruijn
Capture	Cheat	Cheat	Problem	Solved
α -equivalence	Cheat	Cheat	Problem	Solved
Generic subst.	Solved	Solved	Problem	Problem
Malformed syntax	Cheat	Cheat	Problem	Problem

- In **embedded languages** and in **proofs**, HOAS is very common.
- In conventional language implementations and machine-checked formalisations, de Bruijn indices are more popular.
- In your assignments, strings will be used

Evaluating All Approaches

	HOAS		FOAS	
	Proofs	Haskell	Strings	de Bruijn
Capture	Cheat	Cheat	Problem	Solved
α -equivalence	Cheat	Cheat	Problem	Solved
Generic subst.	Solved	Solved	Problem	Problem
Malformed syntax	Cheat	Cheat	Problem	Problem

- In **embedded languages** and in **proofs**, HOAS is very common.
- In conventional language implementations and machine-checked formalisations, de Bruijn indices are more popular.
- In your assignments, strings will be used

Evaluating All Approaches

	HOAS		FOAS	
	Proofs	Haskell	Strings	de Bruijn
Capture	Cheat	Cheat	Problem	Solved
α -equivalence	Cheat	Cheat	Problem	Solved
Generic subst.	Solved	Solved	Problem	Problem
Malformed syntax	Cheat	Cheat	Problem	Problem

- In **embedded languages** and in **proofs**, HOAS is very common.
- In conventional language implementations and machine-checked formalisations, de Bruijn indices are more popular.
- In your assignments, strings will be used 😊



λ-Calculus

The term language we just defined is almost a full featured programming language.

Just change the syntax slightly:

$$\begin{array}{l}
 t ::= \text{Symbol} \quad (\text{symbols}) \\
 \quad | \quad x \quad (\text{variables}) \\
 \quad | \quad t_1 \ t_2 \quad (\text{application}) \\
 \quad | \quad \lambda x. t \quad (\lambda\text{-abstraction})
 \end{array}$$

There is just one rule to evaluate terms, called *β-reduction*:

$$(\lambda x. t) u \mapsto_{\beta} t[x := u]$$

Just as in Haskell, $(\lambda x. t)$ denotes a **function** that, given an argument for x , will return t .

β-reduction

β-reduction is a *congruence*:

$$\frac{}{\overline{(\lambda x. t) u \mapsto_{\beta} t[x := u]}}$$

$$\frac{t \mapsto_{\beta} t'}{s t \mapsto_{\beta} s t'} \quad \frac{s \mapsto_{\beta} s'}{s t \mapsto_{\beta} s' t} \quad \frac{t \mapsto_{\beta} t'}{\lambda x. t \mapsto_{\beta} \lambda x. t'}$$

This means we can pick any reducible subexpression (called a *redex*) and perform β-reduction.

Example:

$$\begin{aligned} (\lambda x. \lambda y. f (y x)) 5 (\lambda x. x) &\mapsto_{\beta} (\lambda y. f (y 5)) (\lambda x. x) \\ &\mapsto_{\beta} f ((\lambda x. x) 5) \\ &\mapsto_{\beta} f 5 \end{aligned}$$

β-reduction

β-reduction is a *congruence*:

$$\frac{}{\overline{(\lambda x. t) u \mapsto_{\beta} t[x := u]}}$$

$$\frac{t \mapsto_{\beta} t'}{s t \mapsto_{\beta} s t'} \quad \frac{s \mapsto_{\beta} s'}{s t \mapsto_{\beta} s' t} \quad \frac{t \mapsto_{\beta} t'}{\lambda x. t \mapsto_{\beta} \lambda x. t'}$$

This means we can pick any reducible subexpression (called a *redex*) and perform β-reduction.

Example:

$$\begin{aligned}
 (\lambda x. \lambda y. f (y x)) 5 (\lambda x. x) &\mapsto_{\beta} (\lambda y. f (y 5)) (\lambda x. x) \\
 &\mapsto_{\beta} f ((\lambda x. x) 5) \\
 &\mapsto_{\beta} f 5
 \end{aligned}$$

β-reduction

β-reduction is a *congruence*:

$$\frac{}{\overline{(\lambda x. t) u \mapsto_{\beta} t[x := u]}}$$

$$\frac{t \mapsto_{\beta} t'}{s t \mapsto_{\beta} s t'} \quad \frac{s \mapsto_{\beta} s'}{s t \mapsto_{\beta} s' t} \quad \frac{t \mapsto_{\beta} t'}{\lambda x. t \mapsto_{\beta} \lambda x. t'}$$

This means we can pick any reducible subexpression (called a *redex*) and perform β-reduction.

Example:

$$\begin{aligned} (\lambda x. \lambda y. f (y x)) 5 (\lambda x. x) &\mapsto_{\beta} (\lambda y. f (y 5)) (\lambda x. x) \\ &\mapsto_{\beta} f ((\lambda x. x) 5) \\ &\mapsto_{\beta} f 5 \end{aligned}$$

β-reduction

β-reduction is a *congruence*:

$$\frac{}{\overline{(\lambda x. t) u \mapsto_{\beta} t[x := u]}}$$

$$\frac{t \mapsto_{\beta} t'}{s t \mapsto_{\beta} s t'} \quad \frac{s \mapsto_{\beta} s'}{s t \mapsto_{\beta} s' t} \quad \frac{t \mapsto_{\beta} t'}{\lambda x. t \mapsto_{\beta} \lambda x. t'}$$

This means we can pick any reducible subexpression (called a *redex*) and perform β-reduction.

Example:

$$\begin{aligned} (\lambda x. \lambda y. f (y x)) 5 (\lambda x. x) &\mapsto_{\beta} (\lambda y. f (y 5)) (\lambda x. x) \\ &\mapsto_{\beta} f ((\lambda x. x) 5) \\ &\mapsto_{\beta} f 5 \end{aligned}$$

β-reduction

β-reduction is a *congruence*:

$$\frac{}{\overline{(\lambda x. t) u \mapsto_{\beta} t[x := u]}}$$

$$\frac{t \mapsto_{\beta} t'}{s t \mapsto_{\beta} s t'} \quad \frac{s \mapsto_{\beta} s'}{s t \mapsto_{\beta} s' t} \quad \frac{t \mapsto_{\beta} t'}{\lambda x. t \mapsto_{\beta} \lambda x. t'}$$

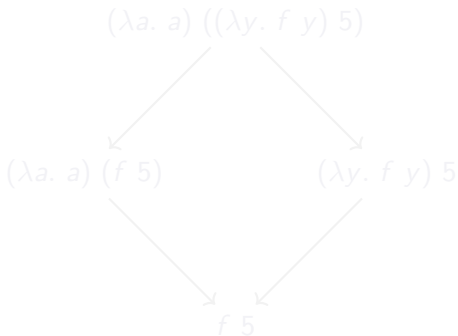
This means we can pick any reducible subexpression (called a *redex*) and perform β-reduction.

Example:

$$\begin{aligned} (\lambda x. \lambda y. f (y x)) 5 (\lambda x. x) &\mapsto_{\beta} (\lambda y. f (y 5)) (\lambda x. x) \\ &\mapsto_{\beta} f ((\lambda x. x) 5) \\ &\mapsto_{\beta} f 5 \end{aligned}$$

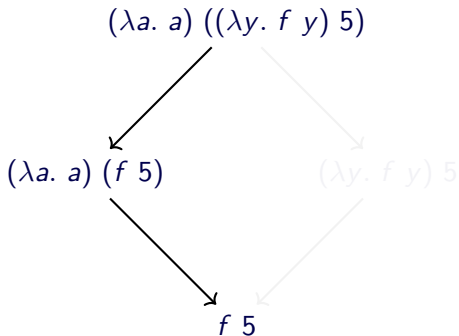
Confluence

Supposing we arrive via one reduction path to an expression that cannot be reduced further (called a *normal form*), then any other reduction path will result in the *same normal form*.



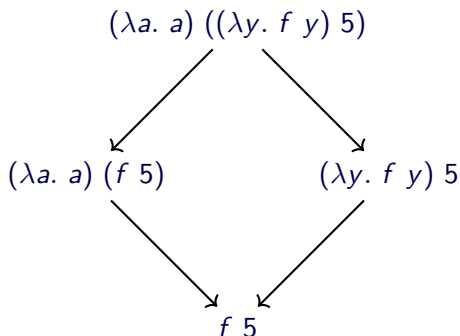
Confluence

Supposing we arrive via one reduction path to an expression that cannot be reduced further (called a *normal form*), then any other reduction path will result in the *same normal form*.



Confluence

Supposing we arrive via one reduction path to an expression that cannot be reduced further (called a *normal form*), then any other reduction path will result in the *same normal form*.



Equivalence

Confluence means we can define another notion of *equivalence*, which equates more than α -equivalence. Two terms are $\alpha\beta$ -equivalent, written $s \equiv_{\alpha\beta} t$ if they β -reduce to α -equivalent normal forms.

η

There is also another equation that cannot be proven from β -equivalence alone, called η -conversion:

$$(\lambda x. f x) \mapsto_{\eta} f$$

Adding this reduction to the system preserves confluence and uniqueness of normal forms, so we have a notion of $\alpha\beta\eta$ -equivalence also.

Equivalence

Confluence means we can define another notion of *equivalence*, which equates more than α -equivalence. Two terms are $\alpha\beta$ -equivalent, written $s \equiv_{\alpha\beta} t$ if they β -reduce to α -equivalent normal forms.

η

There is also another equation that cannot be proven from β -equivalence alone, called η -conversion:

$$(\lambda x. f x) \mapsto_{\eta} f$$

Adding this reduction to the system preserves confluence and uniqueness of normal forms, so we have a notion of $\alpha\beta\eta$ -equivalence also.

Normal Forms

Does every term in λ -calculus have a normal form?

$$(\lambda x. x x)(\lambda x. x x)$$

Try to β -reduce this! (the answer is that it doesn't have a normal form)

Normal Forms

Does every term in λ-calculus have a normal form?

$$(\lambda x. x x)(\lambda x. x x)$$

Try to β-reduce this! (the answer is that it doesn't have a normal form)

Why learn this stuff?

- λ-calculus is a *Turing-complete* programming language. We'll explain how to use it in the extension lecture.
- λ-calculus is the foundation for every functional programming language and some non-functional ones.
- λ-calculus is the foundation of *Higher Order Logic* and *Type Theory*, the two main foundations used for mathematics in interactive proof assistants.
- λ-calculus is the smallest example of a usable programming language, so it's good for teaching about programming languages.