

Simply typed λ -calculus

©Andrey Kruglyak, 2010

Simply typed λ -calculus

- Let us add typing rules to the untyped λ -calculus (extended with integers and operation $+$ and $-$ on integers) that we defined earlier
- Once defined, typing rules can be proved to be sound (progress and preservation properties can be proved to hold), but we will not do that in this course

Syntax

Terms:

$$t ::= x \mid v \mid t \ t \mid t + t \mid t - t$$

Values:

$$v ::= \lambda x:T.t \mid n$$

Types:

$$T ::= \text{Int} \mid T \rightarrow T$$

Typing contexts (only used in typing rules, not part of the concrete syntax of the language):

$$\Gamma ::= \emptyset \mid \Gamma, x:T$$

Typing rules

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{T-Var})$$

$$\frac{}{\emptyset \vdash n : \text{Int}} \quad (\text{T-Int})$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2} \quad (\text{T-Abs})$$

$$\frac{\Gamma \vdash t_1 : \text{Int} \quad \Gamma \vdash t_2 : \text{Int}}{\Gamma \vdash t_1 + t_2 : \text{Int}} \quad (\text{T-Add})$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad (\text{T-App})$$

$$\frac{\Gamma \vdash t_1 : \text{Int} \quad \Gamma \vdash t_2 : \text{Int}}{\Gamma \vdash t_1 - t_2 : \text{Int}} \quad (\text{T-Sub})$$

Evaluation rules

$$\frac{t_2 \rightarrow t_2'}{t_1 t_2 \rightarrow t_1 t_2'} \quad (\text{App1})$$

$$\frac{}{(\lambda x. t_1 : T) v \rightarrow [x \mapsto v] t_1} \quad (\text{App2})$$

$$\frac{t_1 \rightarrow t_1'}{t_1 + t_2 \rightarrow t_1' + t_2} \quad (\text{Add1})$$

$$\frac{t_1 \rightarrow t_1'}{t_1 - t_2 \rightarrow t_1' - t_2} \quad (\text{Sub1})$$

$$\frac{t_2 \rightarrow t_2'}{v + t_2 \rightarrow v + t_2'} \quad (\text{Add2})$$

$$\frac{t_2 \rightarrow t_2'}{v - t_2 \rightarrow v - t_2'} \quad (\text{Sub2})$$

$$\frac{v_1 + v_2 = v_3}{v_1 + v_2 \rightarrow v_3} \quad (\text{Add3})$$

$$\frac{v_1 - v_2 = v_3}{v_1 - v_2 \rightarrow v_3} \quad (\text{Sub3})$$

Can we have polymorphic functions?



Can we have polymorphic functions?

- No! (We don't have type variables in our syntax)

$(\lambda x : \text{Int}. x)$ and $(\lambda x : \text{Int} \rightarrow \text{Int}. x)$ will be two different functions

- This is not the case in polymorphic λ -calculus, where we can
 - allow type variables $(\lambda x : a. x)$, and introduce explicit type abstraction $(\Lambda t. \lambda x : t. x)$ and type application $((\Lambda t. \lambda x : t. x) \text{Int}) \rightarrow (\lambda x : \text{Int}. x)$

Church- vs Curry-style

- Curry-style semantics: meaning (operational semantics in terms of evaluation rules) is assigned to terms regardless of types (type annotations are ignored / can be erased prior to evaluation)

=> Typing rules are used to filter out programs whose behavior we don't like (typical for languages with type inference)

- Church-style semantics: terms and hence evaluation rules explicitly include types, i.e. semantics is only defined for well-typed terms. Then the same term only differing in type notations may mean different things (e.g. $(\lambda x : \text{Int}. x)$ and $(\lambda x : \text{Bool}. x)$ may have different evaluation rules).

=> Typing rules are part of a language definition!
(typical for languages with explicit types)