Confluence

It doesn't matter in what order we do reductions

If t is a λ-term and

$\left.\begin{array}{l} t \rightsquigarrow^* t_1 \\ t \rightsquigarrow^* t_2 \end{array}\right\}$ I can reduce t to two different terms using different reduction sequences

Then there is a term t_3 s.t.

$\left.\begin{array}{l} t_1 \rightsquigarrow^* t_3 \\ t_2 \rightsquigarrow^* t_3 \end{array}\right\}$ I can find a common reduct

In pictures:

$$\begin{tikzcd} & t \arrow[dl, "*"'] \arrow[dr, "*"] & \\ t_1 \arrow[dr, "*"'] & & t_2 \arrow[dl, "*"] \\ & t_3 & \end{tikzcd}$$

Consequence:

Normal Forms are Unique

A term can have only one normal form
(But some terms don't have one)

Evaluation Strategies

for λ-terms

How do we choose which redex to reduce

- **Full β-reduction**

  reduce any redex you want

- **Normal order**

  reduce the leftmost redex (top redex in AST)

  normalizing strategy: if there is a normal form, it will find it

- **Call-by-name**

  - If the term is an application

    $$(f\ u)$$

    reduce f until it becomes an abstraction

    $$f \leadsto^* \lambda x.t$$

    then reduce the main redex

    $$(\lambda x.t)\,u \leadsto t[x := u]$$

  - Never reduce under abstraction

    $\lambda f.t$ is considered a value

Lazy evaluation (Haskell)

The same with term-graphs and sharing

- **Call-by-value**

  In an application

  $$(f\ u)$$

  - reduce f to an abstraction

    $$f \rightsquigarrow^* \lambda x.t$$

  - reduce u to a value

    $$u \rightsquigarrow^* v$$

  Then reduce the main redex

  $$(\lambda x.t)v \rightsquigarrow t[x:=v]$$

Idea:

- in call-by-name the argument u is passed to the function without evaluation: we use its "name", ie its syntactic expression
- in call-by-value we evaluate the argument to a value before passing it to the function

Type Systems

Reasons to introduce types

- **Readability**
  Human users understand programs better when they have types
- **Correctness**
  Types ensure some safety for the input-output relation of programs
- **Outlaw meaningless terms**
  Operators can be applied only to arguments of the right type
- **Efficient Compilation**
  Operations on a specific data type can be optimized

The need for types

The term $\lambda x.\lambda y.y$ has many meanings:

- Second Projection
- Boolean Value false
- Natural number zero
  (by α-eq.: $\lambda f.\lambda x.x$)

We want to distinguish the different uses

General Form of Typing Rules

Two kinds of expressions

terms denote values/objects

types denote sets of values

Typing assertion:

$$t : T$$

the term t belongs to the type T

Variables should also have a type

context: assigment of types to variables

$$\underbrace{x:A,\ y:B,\ z:C}_{\Gamma}$$

we use capital Greek letters for contexts

The type of a term t depends on the type of its free variables

Typing Judgment: $\Gamma \vdash t : T$

Assuming x has type A, y has type B, z has type C we can derive that t has type T

General form of typing rules:

$$\frac{\Gamma_1 \vdash t_1 : T_1 \quad \cdots \quad \Gamma_n \vdash t_n : T_n}{\Gamma_0 \vdash t_0 : T_0}$$

Assume the terms $t_1, \ldots, t_n$ have these types

Then t_0 has type T_0

(Judgments can have different contexts)

If all judgments have the same context, we can leave it out

$$\frac{t_1 : T_1 \quad \dots \quad t_n : T_n}{t_0 : T_0}$$

Example: Type system for simple Arithmetic Expressions

Only two types: $T ::= Nat \mid Bool$

Typing Rules:

No Assumptions

$$\frac{}{true : Bool} \qquad \frac{}{false : Bool} \qquad \frac{}{zero : Nat}$$

$$\frac{t : Nat}{succ\ t : Nat} \qquad \frac{t : Nat}{pred\ t : Nat}$$

$$\frac{t : Nat}{iszero\ t : Bool}$$

$$\frac{t_1 : Bool \quad t_2 : T \quad t_3 : T}{if\ t_1\ then\ t_2\ else\ t_3 : T}$$

any type, but the same for both branches

Meaningless terms cannot be typed:

if (pred false) then (succ true) else zero

does not have a type

Full derivation of a typing judgment

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{zero} : \texttt{Nat}}}{\texttt{succ zero} : \texttt{Nat}}}{\texttt{iszero (succ zero)} : \texttt{Bool}} \qquad \dfrac{}{\texttt{zero} : \texttt{Nat}} \qquad \dfrac{\dfrac{}{\texttt{zero} : \texttt{Nat}}}{\texttt{pred zero} : \texttt{Nat}}}{\texttt{if (iszero(succ zero)) then zero else (pred zero)} : \texttt{Nat}}$$

In this system there are no variables

no need for contexts