MTAT.05.105 Type Theory

Untyped λ-calculus

Syntax of λ-calculus

- Assume a countable set of variables.
- Syntax of λ-terms in BNF:

$$\begin{array}{lcll} e & ::= & x & \text{variable} \\ & | & (e_1\ e_2) & \text{application} \\ & | & (\lambda x.\ e) & \text{abstraction} \end{array}$$

- Bracketing conventions:

$$\begin{array}{lcl} e_1\ e_2\ \ldots\ e_n & \equiv & ((\ldots(e_1\ e_2)\ldots)e_n) \\ \lambda x.\ e_1\ e_2\ \ldots\ e_n & \equiv & (\lambda x.\ (e_1\ e_2\ \ldots\ e_n)) \\ \lambda x_1\ x_2\ \ldots\ x_n.\ e & \equiv & (\lambda x_1.\ (\lambda x_2.\ (\ldots(\lambda x_n.\ e)\ldots)) \end{array}$$

- Examples:

$$\lambda x.\ x \qquad\qquad ((\lambda x.\ (\lambda f.\ f\ x))\ y)\,(\lambda z.\ z)$$

Syntax of λ-calculus

- Note: pure λ-calculus "talks" only about function.
- There are no numbers or other data types.
- They are not needed, as they can be expressed as λ-terms.
- However, for convenience, we often use numbers and arithmetic/logic operations, as they would be "built-in".
- We also use macro-definitions (must be non-recursive).
- Examples:

$$\begin{array}{lcl} \mathbf{add} & \equiv & \lambda x\, y.\; x + y \\ \mathbf{dbl} & \equiv & \lambda x.\; 2 * x \\ \mathbf{I} & \equiv & \lambda x.\; x \\ \mathbf{K} & \equiv & \lambda x\, y.\; x \\ \mathbf{S} & \equiv & \lambda f\, g\, x.\; f\, x\, (g\, x) \end{array}$$

Free and bound variables

- An occurence of a variable is a binding occurence if it is defined by a lambda.
- An occurence is bound if it is in the scope of a binding occurrence with the same name.
- Other occurrences are free.

$$(\lambda x.\ \underset{\text{free}}{y}\ \underset{\text{bound}}{x})\ (\lambda y.\ \underset{\text{free}}{x}\ \underset{\text{bound}}{y})$$

Free and bound variables

- Free variables are defined by induction:

$$\begin{array}{lcl} \mathrm{FV}(x) & = & \{x\} \\ \mathrm{FV}(e_1\ e_2) & = & \mathrm{FV}(e_1) \cup \mathrm{FV}(e_2) \\ \mathrm{FV}(\lambda x.\ e) & = & \mathrm{FV}(e)\ -\ \{x\} \end{array}$$

- λ-terms without free variables are closed.
- Examples:

$$\begin{array}{lcl} \mathrm{FV}(\lambda x\, y.\ x\ y) & = & \emptyset \\ \mathrm{FV}(\lambda x.\ (\lambda y.x)\,(\lambda z.y)) & = & \{z\} \end{array}$$

α-conversion

- Names of bound variables do not matter!
- λ-terms e_1 and e_2 are α-congruent (denoted by $e_1 =_\alpha e_2$) if they are identical up to renaming of bound variables.
- Examples:

$$
\begin{array}{lll}
\lambda x.\ x & =_\alpha & \lambda y.\ y \\
\lambda x.\ f\ x & =_\alpha & \lambda z.\ f\ z \\
\lambda x.\ (\lambda y.\ y)\ x & =_\alpha & \lambda y.\ (\lambda y.\ y)\ y \\
\lambda x\ y.\ x + y & \neq_\alpha & \lambda y\ y.\ y + y
\end{array}
$$

Substitution

- The fundamental principle of computation in λ-calculus is replacement of formal by actual parameters.
- To evaluate an application $(\lambda x.\, e_1)\, e_2$, substitute e_2 for x in e_1.
- Substitution is denoted by $e_1[x \mapsto e_2]$.
- Must avoid variable capture.
- Examples:

$$
\begin{array}{rcl}
(\lambda x.\ y\, x)[y \mapsto \lambda z.\, z] & = & \lambda x.\ (\lambda z.\, z)\, x \\
(\lambda x.\ y\, x)[x \mapsto \lambda z.\, z] & = & \lambda x.\ y\, x \\
(\lambda x.\ y\, x)[y \mapsto \lambda z.\, x] & \neq & \lambda x.\ (\lambda z.\, x)\, x
\end{array}
$$

Substitution

Definition of capture avoiding substitution:

$$y[x \mapsto e] = \begin{cases} e & \text{if } x = y \\ y & \text{otherwise} \end{cases}$$

$$(e_1\, e_2)[x \mapsto e] = (e_1[x \mapsto e])\ (e_2[x \mapsto e])$$

$$(\lambda y.\, e_1)[x \mapsto e] = \begin{cases} \lambda y.\, e_1 & \text{if } x = y \\ \lambda y.\, e_1[x \mapsto e] & \text{if } y \notin \mathrm{FV}(e) \\ \lambda z.\, e_1[y \mapsto z][x \mapsto e] & \text{otherwise} \end{cases}$$

β-reduction

- The evaluation of λ-terms is specified by a repeated application of reduction rules.
- β-reduction rule:

$$(\lambda x.\, e_1)\, e_2 \quad \to_\beta \quad e_1[x \mapsto e_2]$$

- A subexpression in a form $(\lambda x.\, e_1)\, e_2$ is called a (β-)redex.
- Note: a λ-term may any number redexes.
- A λ-term without any (β-)redexes is in (β-)normal form.
- Examples:

$\lambda x.\, x\, (\lambda y.\, x)$	no redexes (ie. normal form)
$\lambda x.\, \underline{(\lambda y.\, y)\, 3}$	a single redex
$\lambda f.\, f\, \underline{((\lambda x.\, x)\, 3)}\ \underline{((\lambda x.\, x)\, 4)}$	two redexes
$\underline{(\lambda f.\, f\, \underline{((\lambda x.\, x)\, 3)})\ (\lambda x.\, x)}$	two (overlapping) redexes

β-reduction

Single-step β-reduction:

$$\frac{}{(\lambda x.\, e_1)\, e_2 \rightarrow_\beta e_1[x \mapsto e_2]}$$

$$\frac{e_1 \rightarrow_\beta e_2}{e_1\, e_0 \rightarrow_\beta e_2\, e_0} \qquad \frac{e_1 \rightarrow_\beta e_2}{e_0\, e_1 \rightarrow_\beta e_0\, e_2}$$

$$\frac{e_1 \rightarrow_\beta e_2}{\lambda x.e_1 \rightarrow_\beta \lambda x.e_2}$$

β-reduction

- Multi-step β-reduction:

$$\frac{e_1 \to_\beta e_2}{e_1 \twoheadrightarrow_\beta e_2} \qquad \frac{}{e \twoheadrightarrow_\beta e} \qquad \frac{e_1 \twoheadrightarrow_\beta e_2 \quad e_2 \twoheadrightarrow_\beta e_3}{e_1 \twoheadrightarrow_\beta e_3}$$

- Note: β-reduction, as defined, is highly non-deterministic.
- Doesn't determine which redex to reduce next.
- Example:

$$\begin{array}{lll}
\underline{(\lambda f.\, f\,((\lambda x.\, x)\, 3))\ (\lambda x.\, x)} & \to_\beta & \underline{(\lambda x.\, x)\,((\lambda x.\, x)\, 3))} \\
 & \to_\beta & \underline{(\lambda x.\, x)\, 3} \\
 & \to_\beta & 3 \\
(\lambda f.\, f\,\underline{((\lambda x.\, x)\, 3)})\ (\lambda x.\, x) & \to_\beta & \underline{(\lambda f.\, f\ 3)\ (\lambda x.\, x)} \\
 & \to_\beta & \underline{(\lambda x.\, x)\, 3} \\
 & \to_\beta & 3
\end{array}$$

Reduction orders

- The reduction order doesn't matter!
- Church-Rosser theorem: for any λ-terms e_0, e_1 and e_2, if $e_0 \twoheadrightarrow_\beta e_1$ and $e_0 \twoheadrightarrow_\beta e_2$ then there exists e_3 such that $e_1 \twoheadrightarrow_\beta e_3$ and $e_2 \twoheadrightarrow_\beta e_3$.
- Corollary: if a λ-term has a normal form, the normal form is unique.
- Note: there exist λ-terms without a normal form.
- Example:

$$
\begin{array}{rcl}
(\lambda x.\, x\, x)\,(\lambda x.\, x\, x) & \twoheadrightarrow_\beta & (\lambda x.\, x\, x)\,(\lambda x.\, x\, x) \\
& \twoheadrightarrow_\beta & (\lambda x.\, x\, x)\,(\lambda x.\, x\, x) \\
& \twoheadrightarrow_\beta & (\lambda x.\, x\, x)\,(\lambda x.\, x\, x) \\
& \twoheadrightarrow_\beta & (\lambda x.\, x\, x)\,(\lambda x.\, x\, x) \\
& \ldots &
\end{array}
$$

Reduction orders

- The reduction order does matter!
- Normal order: always reduce a leftmost-outermost redex.

$$\underline{(\lambda x.\, y)((\lambda x.\, x\, x)(\lambda x.\, x\, x))} \twoheadrightarrow_\beta y$$

- Applicative order: always reduce a leftmost-innermost redex.

$$\begin{aligned} & (\lambda x.\, y)(\underline{(\lambda x.\, x\, x)(\lambda x.\, x\, x)}) \\ \twoheadrightarrow_\beta \quad & (\lambda x.\, y)(\underline{(\lambda x.\, x\, x)(\lambda x.\, x\, x)}) \\ & \ldots \end{aligned}$$

- Normalization Theorem: the normal order reduction sequence reaches a normal form whenever it exists for a given λ-term.

Weak head normal forms

- Evaluation inside the function bodies (ie. under lambdas) is difficult to implement efficiently.
- Thefore, usually a weaker notion of normal forms is used.
- A λ-term e is in weak head normal form if it is in a form

$$e \equiv \begin{cases} x\, e_1 \, \ldots \, e_m & m \geq 0 \\ \lambda x.\, e_1 & \end{cases}$$

- Aside: a λ-term e is in head normal form if

$$e \equiv \lambda x_1 \ldots x_n.\, x\, e_1 \, \ldots \, e_m \qquad n, m \geq 0$$

Big-step semantics

- β-reduction with some specified reduction order gives a low-level view of evaluation (small-step semantics).
- While sufficient for reasoning about evaluation, it's not very good for automated evaluation.
- Natural semantics (also called big-step semantics) defines a *procedure* for program evaluation.
- Use v to represent a value (a λ-term in WHNF).
- Notation $e \downarrow v$ denotes "e evaluates to v".

Big-step semantics

- Evaluation rules:

$$\frac{}{x \downarrow x}\ var \qquad\qquad \frac{}{\lambda x.e \downarrow \lambda x.e}\ abs$$

$$\frac{e_1 \downarrow \lambda x.\, e_3 \quad e_2 \downarrow v_2 \quad e_3[x \mapsto v_2] \downarrow v_3}{(e_1\ e_2) \downarrow v_3}\ appE$$

- The rule *appE* corresponds to the eager evaluation (applicative order reduction).
- The rule corresponding to the normal order reduction:

$$\frac{e_1 \downarrow \lambda x.\, e_3 \quad e_3[x \mapsto e_2] \downarrow v}{(e_1\ e_2) \downarrow v}\ appL$$

Soundness of natural semantics

- Theorem: If $e \downarrow v$ then $e \twoheadrightarrow_\beta v$.
- Proof by induction over the structure of the proof tree.
- Base cases are trivial:
  - If $\dfrac{}{x \downarrow x}\ var$ then $x \twoheadrightarrow_\beta x$.
  - If $\dfrac{}{\lambda x.e \downarrow \lambda x.e}\ abs$ then $\lambda x.e \twoheadrightarrow_\beta \lambda x.e$.
- Otherwise, suppose the last rule was for an application:

$$\frac{e_1 \downarrow \lambda x.\, e_3 \quad e_3[x \mapsto e_2] \downarrow v}{(e_1\, e_2) \downarrow v}\ appL$$

Chain these together:

$$\begin{array}{lll} & e_1\, e_2 & \text{initial term} \\ \twoheadrightarrow_\beta & (\lambda x.\, e_3)\, e_2 & \text{by ind. hyp.} \\ \twoheadrightarrow_\beta & e_3[x \mapsto e_2] & \beta\text{-reduction} \\ \twoheadrightarrow_\beta & v & \text{by ind. hyp.} \end{array}$$

Programming in λ-calculus

- λ-calculus is a Turing complete programming language.
- Church thesis: Every computable function is representable in pure λ-calculus.
- Booleans:

$$
\begin{array}{lcll}
\mathbf{true} & \equiv & \lambda xy.\ x & (\equiv\ \mathbf{K}) \\
\mathbf{false} & \equiv & \lambda xy.\ y & \\
\mathbf{cond} & \equiv & \lambda t.\ t\ \mathbf{true}\ \mathbf{false} &
\end{array}
$$

- Natural numbers (Church numerals):

$$
\begin{array}{lcl}
\underline{n} & \equiv & \lambda f\ x.\ f^n\ x \\
\mathbf{succ} & \equiv & \lambda n.\ \lambda f\ x.\ n\ f\ (f\ x) \\
\mathbf{iszero} & \equiv & \lambda n.\ n\ (\lambda x.\ \mathbf{false})\ \mathbf{true} \\
\mathbf{add} & \equiv & \lambda m\ n.\ \lambda f\ x.\ m\ f\ (n\ f\ x)
\end{array}
$$

Programming in λ-calculus

- Curry's paradoxial combinator:

$$\mathbf{Y} \equiv \lambda f.(\lambda x.f(x\,x))\,(\lambda x.f(x\,x))$$

- Combinator $\mathbf{Y}$ is a fixed point combinator:

$$\begin{array}{rcl} \mathbf{Y}\,e & \twoheadrightarrow_\beta & (\lambda x.e(x\,x))\,(\lambda x.e(x\,x)) \\ & \twoheadrightarrow_\beta & e((\lambda x.e(x\,x))\,(\lambda x.e(x\,x))) \\ & = & e\,(\mathbf{Y}\,e) \end{array}$$

- Fixed point combinators can be used to define recursive functions.
- Example:

$$\mathbf{fact} \equiv \mathbf{Y}\ \lambda f.\,\lambda n.\,\mathbf{if}\,(n = 0)\,\mathbf{then}\,1\,\mathbf{else}\,n * f\,(n-1)$$