Lambda Calculus

Tobias Nipkow

November 27, 2023

Contents

Chapter 1

Untyped Lambda Calculus

1.1 Syntax

1.1.1 Terms

Definition 1.1.1. The set of lambda calculus **terms** is defined as follows:

$$t \quad ::= \quad c \quad | \quad x \quad | \quad (t_1\ t_2) \quad | \quad (\lambda x.\, t)$$

$(t_1\ t_2)$ is called **application** and represents the application of a function t_1 to an argument t_2.

$(\lambda x.\, t)$ is called **abstraction** and represents the function with formal parameter x and body t; x is bound in t.

Convention:

x, y, z	variables
c, d, f, g, h	constants
a, b	atoms = variables $\cup$ constants
r, s, t, u, v, w	terms

In lambda calculus there is one computation rule called β-reduction: $((\lambda x.\, s)\ t)$ can be reduced to $s[t/x]$, the result of replacing the arguments t for the formal parameter x in s. Examples:

$$\begin{array}{rcl} ((\lambda x.\, ((f\, x)\, x))\, 5) & \to_\beta & ((f\, 5)\, 5) \\ ((\lambda x.\, x)\, (\lambda x.\, x)) & \to_\beta & (\lambda x.\, x) \\ (x\, (\lambda y.\, y)) & \text{cannot be reduced} & \end{array}$$

The precise definition of $s[t/x]$ needs some work.

Notation:

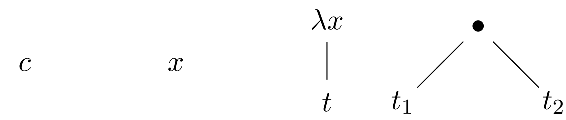
- Application associates to the left: $(t_1 \ldots t_n) \equiv (((t_1\ t_2)t_3) \ldots t_n)$
- Outermost parentheses are omitted: $t_1 \ldots t_n \equiv (t_1 \ldots t_n)$
- λ binds to the right as far as possible.

  Example: $\lambda x.\, x\, x \equiv \lambda x.\, (x\, x) \not\equiv (\lambda x.\, x)\, x$
- Consecutive λs can be combined: $\lambda x_1 \ldots x_n.\, s \equiv \lambda x_1.\, \ldots \lambda x_n.\, s$

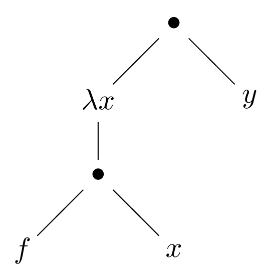
Terms as trees:

term: c $\quad x \quad (\lambda x.\, t) \quad (t_1\, t_2)$

tree:



Example: term to tree $(\lambda x.\, f\, x)\, y$



Definition 1.1.2. Term s is **subterm** of t, if the tree corresponding to s is a subtree of the tree corresponding to t. Term s is a **proper subterm** of t if s is a subterm of t and $s \neq t$.

Example:

Is $s\ (t\ u)$ a subterm of $r\ s\ (t\ u)$?
No, $\quad r\ s\ (t\ u) \equiv (r\ s)\ (t\ u)$

1.1.2 Currying (Schönfinkeln)

Currying means reducing a function with multiple arguments to functions with a single argument.

Example:

$$f : \begin{cases} \mathbb{N} \to \mathbb{N} \\ x \mapsto x + x \end{cases}$$

In lambda calculus: $f = \lambda x.\, x + x$

$$g : \begin{cases} \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\ (x, y) \mapsto x + y \end{cases}$$

Incorrect translation of g: $\quad \lambda(x, y).\, x + y$
Not permitted by lambda calculus syntax!

Instead: $\quad g \quad \cong \quad g' \quad = \quad \lambda x.\, \lambda y.\, x + y$
Therefore: $\quad g'\colon\ \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$

Example of evaluation: $\quad g(5, 3) = 5 + 3$

Evaluation in lambda-calculus:

$$
\begin{array}{lclcl}
g'\ 5\ 3 & \equiv & ((g'\ 5)\ 3) & \equiv & (((\lambda x.\, \lambda y.\, x + y)\ 5)\ 3) \\
 & & & \to_\beta & ((\lambda y.\, 5 + y)\ 3) \\
 & & & \to_\beta & 5 + 3
\end{array}
$$

The term g' 5 is well defined. This is called *partial application*.
Illustration: In the table for g

g	1	2	$\cdots$
1	$\cdot$	$\cdot$	$\cdots$
2	$\cdot$	$\cdot$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

g' 5 corresponds to the unary function that is given by row 5 of the table.

In set theory: $\quad (A \times B) \to C \quad \cong \quad A \to (B \to C)$
(”‘$\cong$”’: isomorphism in set theory)

1.1.3 Static binding and substitution

A variable x in term s is **bound** by the first λx above x (when viewing the term as a tree). If there is no λx above some x, that x is called **free** in s.

Example:

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

free variable

Each arrow points from the occurence of a variable to the binding λ.

The set of free variables of a term can be defined recursively:

$$
\begin{array}{lcl}
FV\colon \quad \text{term} & \to & \text{set of variables} \\
FV(c) & = & \emptyset \\
FV(x) & = & \{x\} \\
FV(s\ t) & = & FV(s) \cup FV(t) \\
FV(\lambda x.\, t) & = & FV(t) \setminus \{x\}
\end{array}
$$

Definition 1.1.3. A term t is said to be **closed** if $FV(t) = \emptyset$.

Definition 1.1.4. The **substitution** of t for all free occurrences of x in s (pronounced "s with t for x") is recursively defined:

$$
\begin{array}{rcll}
x[t/x] & = & t & \\
a[t/x] & = & a & \text{if } a \neq x \\
(s_1\ s_2)[t/x] & = & (s_1[t/x])\ (s_2[t/x]) & \\
(\lambda x.\, s)[t/x] & = & \lambda x.\, s & \\
(\lambda y.\, s)[t/x] & = & \lambda y.\, (s[t/x]) & \text{if } x \neq y \wedge y \notin FV(t) \\
(\lambda y.\, s)[t/x] & = & \lambda z.\, (s[z/y][t/x]) & \text{if } x \neq y \wedge z \notin FV(s) \cup FV(t) \cup \{x\}
\end{array}
$$

To make the choice of z in the last rule deterministic, assume that the variables are linearly ordered and that we take the *first* z such that $z \notin FV(t) \cup FV(s) \cup \{x\}$. The next to last equation is an optimized form of the last equation that avoids unnecessary renamings.

Example:

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

Lemma 1.1.5.

$$\begin{array}{lcll} s[x/x] & = & s & \\ s[t/x] & = & s & \textit{if } x \notin FV(s) \\ s[y/x][t/y] & = & s[t/x] & \textit{if } y \notin FV(s) \\ s[t/x][u/y] & = & s[u/y][t[u/y]/x] & \textit{if } x \neq y \wedge x \notin FV(u) \\ s[t/x][u/y] & = & s[u/y][t/x] & \textit{if } x \neq y \wedge y \notin FV(t) \wedge x \notin FV(u) \end{array}$$

Remark: Some of these equations hold only up to renaming of bound variables. For example, take equation 3 with $s = \lambda y.\, x$ and $t = z$: $s[y/x][t/y] = (\lambda y.\, x)[y/x][z/y] = (\lambda y'.\, y)[z/y] = \lambda y'.\, z$ but $s[t/x] = \lambda y.\, z$. We will identify terms like $\lambda y'.\, z$ and $\lambda y.\, z$ below.

1.1.4 α-conversion

If s and t are identical up to renaming of bound variables we write $s =_\alpha t$. Motto:

Gebundene Namen sind Schall und Rauch.

Example:

$$\begin{array}{rcl} x\ (\lambda x\, y.\, x\ y) & =_\alpha & x\ (\lambda y\, x.\, y\ x) \quad =_\alpha \quad x\ (\lambda z\, y.\, z\ y) \\ & \neq_\alpha & z\ (\lambda z\, y.\, z\ y) \\ & \neq_\alpha & x\ (\lambda x\, x.\, x\ x) \end{array}$$

Definition 1.1.6.

$$\frac{}{a =_\alpha a} \qquad \frac{s_1 =_\alpha t_1 \quad s_2 =_\alpha t_2}{(s_1\ s_2) =_\alpha (t_1\ t_2)} \qquad \frac{z \notin V(s) \cup V(t) \quad s[x := z] =_\alpha t[y := z]}{(\lambda x.\, s) =_\alpha (\lambda y.\, t)}$$

where $V(t)$ is the set of all variables in t:

$$V(c) = \emptyset, \qquad V(x) = \{x\}, \qquad V(s\ t) = V(s) \cup V(t), \qquad V(\lambda x.\, t) = V(t) \cup \{x\}$$

and $s[x := t]$ is non-renaming substitution:

$$\begin{array}{rcll} x[x := t] & = & t & \\ a[x := t] & = & a & \text{if } a \neq x \\ (s_1\ s_2)[x := t] & = & (s_1[x := t]\ s_2[x := t]) & \\ (\lambda x.\, s)[x := t] & = & (\lambda x.\, s) & \\ (\lambda y.\, s)[x := t] & = & (\lambda y.\, s[x := t]) & \text{if } x \neq y \end{array}$$

Convention:

1. We identify α-equivalent terms, i.e. we work with α-equivalent classes of terms. Example: $\lambda x.\, x = \lambda y.\, y$.

$$\begin{array}{ccccc} & \nearrow_\beta & (\lambda x.\, x\ x)z & \searrow_\beta & \\ (\lambda x.\, x\ x)((\lambda y.\, y)z) & & & & z\ z \\ & \searrow_\beta & ((\lambda y.\, y)z)((\lambda y.\, y)z) & \overset{2}{\nearrow}_\beta & \end{array}$$

Figure 1.1: $\rightarrow_\beta$ is confluent?

2. Bound variables are automatically renamed in such a way that they are different from all the free variables. Example: Let $K = \lambda x.\, \lambda y.\, x$:

$$\begin{array}{llll} K\ s & \rightarrow_\beta & \lambda y.\, s & (\text{if } y \notin FV(s)) \\ K\ y & \rightarrow_\beta & \lambda y'.\, y & (y \text{ is free in } y \text{ and that's why } y \text{ is renamed as } y') \end{array}$$

This simplifies substitution: if $x \neq y$ then

$$(\lambda y.\, s)[t/x] = \lambda y.\, (s[t/x])$$

because by automatic renaming $y \notin FV(t)$.

1.2 β-reduction (contraction)

Definition 1.2.1. A **β-redex** (**red**ucible **ex**pression) is a term of form $(\lambda x.\, s)t$. We define **β-reduction** by

$$C[(\lambda x.\, s)t] \quad \rightarrow_\beta \quad C[s[t/x]]$$

Here $C[v]$ is a term with a subterm v, and C is a context, i.e. a term with a hole where v is put.

A term t is in **β-normal form** if it is in normal form with regard to $\rightarrow_\beta$.

The **reflexive transitive closure** of $\rightarrow_\beta$ is denoted by $\rightarrow_\beta^*$.

Example: $\lambda x.\, \underbrace{(\lambda x.\, x\ x)(\lambda x.\, x)} \rightarrow_\beta \lambda x.\, \underbrace{(\lambda x.\, x)(\lambda x.\, x)} \rightarrow_\beta \lambda x.\, \lambda x.\, x$

Note:

- A term may have more than one β-reduct. Example: see Fig. 1.1.
- β-reduction may not terminate. Example: $\Omega := (\lambda x.\, x\ x)(\lambda x.\, x\ x) \rightarrow_\beta \Omega \rightarrow_\beta \Omega \rightarrow_\beta \ldots$.

Definition 1.2.2. Alternative to definition 1.2.1 one can define $\rightarrow_\beta$ inductively as follows:

1. $(\lambda x.\, s)t \rightarrow_\beta s[t/x]$
2. $s \rightarrow_\beta s' \quad \Rightarrow \quad (s\ t) \rightarrow_\beta (s'\ t)$
3. $s \rightarrow_\beta s' \quad \Rightarrow \quad (t\ s) \rightarrow_\beta (t\ s')$
4. $s \rightarrow_\beta s' \quad \Rightarrow \quad \lambda x.\, s \rightarrow_\beta \lambda x.\, s'$

That is to say, $\rightarrow_\beta$ is the smallest relation that contains the above-mentioned four rules.

Lemma 1.2.3. *If $s \rightarrow_\beta^* s'$ then $\lambda x.\, s \rightarrow_\beta^* \lambda x.\, s'$, $(s\ t) \rightarrow_\beta^* (s'\ t)$ and $(t\ s) \rightarrow_\beta^* (t\ s')$.*

Proof by induction on the length of the sequence $s \rightarrow_\beta^* s'$.

Lemma 1.2.4. $t \rightarrow_\beta^* t' \;\Rightarrow\; s[t/x] \rightarrow_\beta^* s[t'/x]$

Proof: by induction on s:

1. $s = x$: obvious
2. $s = y \neq x$: $s[t/x] = y \rightarrow_\beta^* y = s[t'/x]$
3. $s = c$: as in 2.
4. $s = (s_1\ s_2)$:
   $$\begin{aligned}(s_1\ s_2)[t/x] &= (s_1[t/x])\,(s_2[t/x]) \rightarrow_\beta^* (s_1[t'/x])\,(s_2[t/x]) \rightarrow_\beta^* \\ &\rightarrow_\beta^* (s_1[t'/x])\,(s_2[t'/x]) = (s_1\ s_2)[t'/x] = s[t'/x]\end{aligned}$$
   (using the induction hypothesis $s_i[t/x] \rightarrow_\beta^* s_i[t'/x]$, $i = 1, 2$, as well as transitivity of $\rightarrow_\beta^*$)
5. $s = \lambda y.\, r$: $s[t/x] = \lambda y.\,(r[t/x]) \rightarrow_\beta^* \lambda y.\,(r[t'/x]) = (\lambda y.\, r)[t'/x] = s[t'/x]$
   (using the induction hypothesis $r[t/x] \rightarrow_\beta^* r[t'/x]$) $\square$

Lemma 1.2.5. $s \rightarrow_\beta s' \;\Rightarrow\; s[t/x] \rightarrow_\beta s'[t/x]$

Proof: by induction on the derivation of $s \rightarrow_\beta s'$ (rule induction) as defined in Definition 1.2.2.

1. $s = (\lambda y.\, r)u \rightarrow_\beta r[u/y] = s'$:
   $s[t/x] = (\lambda y.\,(r[t/x]))(u[t/x]) \rightarrow_\beta (r[t/x])[u[t/x]/y] = (r[u/y])[t/x] = s'[t/x]$
2. $s_1 \rightarrow_\beta s_1'$ and $s = (s_1\ s_2) \rightarrow_\beta (s_1'\ s_2) = s'$:
   Induction hypothesis: $s_1[t/x] \rightarrow_\beta s_1'[t/x]$
   $\Rightarrow\; s[t/x] = (s_1[t/x])(s_2[t/x]) \rightarrow_\beta (s_1'[t/x])(s_2[t/x]) = (s_1'\ s_2)[t/x] = s'[t/x]$
3. Analogous to 2.
4. Exercise. $\square$

Corollary 1.2.6. $s \rightarrow_\beta^n s' \;\Rightarrow\; s[t/x] \rightarrow_\beta^n s'[t/x]$

Proof: by induction on n $\square$

Corollary 1.2.7. $s \xrightarrow{*}_\beta s' \wedge t \xrightarrow{*}_\beta t' \quad\Rightarrow\quad s[t/x] \xrightarrow{*}_\beta s'[t'/x]$

Proof: $s[t/x] \xrightarrow{*}_\beta s'[t/x] \xrightarrow{*}_\beta s'[t'/x]$

Does this also hold? $t \rightarrow_\beta t' \;\Rightarrow\; s[t/x] \rightarrow_\beta s[t'/x]$

Exercise 1.2.8. Show $s \rightarrow_\beta t \;\Rightarrow\; FV(s) \supseteq FV(t)$. Why does $FV(s) = FV(t)$ not hold?

1.2.1 Confluence

We try to prove confluence via the diamond property. As seen in Fig 1.1, $\rightarrow_\beta$ does not have the diamond property. There $t := ((\lambda y.\, y)z)((\lambda y.\, y)z)$ cannot be reduced to $z\ z$ in one step.

1. Attempt: parallel reduction of independent redexes (as symbol: $\rightrightarrows$) since $t \rightrightarrows z\ z$.

Problem: $\rightrightarrows$ does not have the diamond property either:

$$
\begin{tikzcd}
 & ((\lambda x.\, x\ c)\ d) \arrow[dr, shift left=0.5] \arrow[dr, shift right=0.5] & \\
\underbrace{(\lambda x.\, \overbrace{(\lambda y.\, x\ y)\ c})(\overbrace{(\lambda x.\, x)\ d})} \arrow[ur, shift left=0.5] \arrow[ur, shift right=0.5] \arrow[dr, shift left=0.5] \arrow[dr, shift right=0.5] & & d\ c \\
 & \underbrace{(\lambda y.\ \underbrace{((\lambda x.\, x)\ d)}\ y)\ c} &
\end{tikzcd}
$$

$(\lambda y.\, ((\lambda x.\, x)d)y)c \rightrightarrows c\ d$ does *not* hold since $(\lambda y.\, ((\lambda x.\, x)d)y)c$ contains *nested* redexes.

Definition 1.2.9. The parallel (and nested) reduction relation $>$ is defined inductively:

1. $s > s$
2. $\lambda x.\, s > \lambda x.\, s'$ if $s > s'$
3. $(s\ t) > (s'\ t')$ if $s > s'$ and $t > t'$ (parallel)
4. $(\lambda x.\, s)t > s'[t'/x]$ if $s > s'$ and $t > t'$ (parallel and nested)

Example:

$$\underbrace{(\lambda x.\, (\underbrace{(\lambda y.\, y)\ x}_{x}))(\underbrace{(\lambda x.\, x)\ z}_{z})} > z$$

Note:

> $>$ is proper subset of $\rightarrow_\beta^*$: $(\lambda f.\, f\ z)(\lambda x.\, x) \rightarrow_\beta (\lambda x.\, x)z \rightarrow_\beta z$ and $(\lambda f.\, f\ z)(\lambda x.\, x) > (\lambda x.\, x)z$ hold, but $(\lambda f.\, f\ z)(\lambda x.\, x) > z$ does not.

Lemma 1.2.10. $s \rightarrow_\beta t \Rightarrow s > t$

Proof: by induction on the derivation of $s \rightarrow_\beta t$ according to definition 1.2.2.

1. If: $s = (\lambda x.\, u)\ v \rightarrow_\beta u[v/x] = t$
$\Rightarrow (\lambda x.\, u)\ v > u[v/x] = t$, since $u > u$ and $v > v$

Remaining cases: exercises □

Lemma 1.2.11. $s > t \Rightarrow s \rightarrow_\beta^* t$

Proof: by induction on the derivation of $s > t$ according to definition 1.2.9.

4. If: $s = (\lambda x.\, u)\ v > u'[v'/x] = t, u > u', v > v'$
Induction hypotheses: $u \xrightarrow{*} u', v \xrightarrow{*} v'$
$s = (\lambda x.\, u)v \rightarrow_\beta^* (\lambda x.\, u')v \rightarrow_\beta^* (\lambda x.\, u')v' \rightarrow_\beta u'[v'/x]$
Remaining cases: left as exercise □

Therefore $\xrightarrow{*}_\beta$ and $>^*$ are identical.

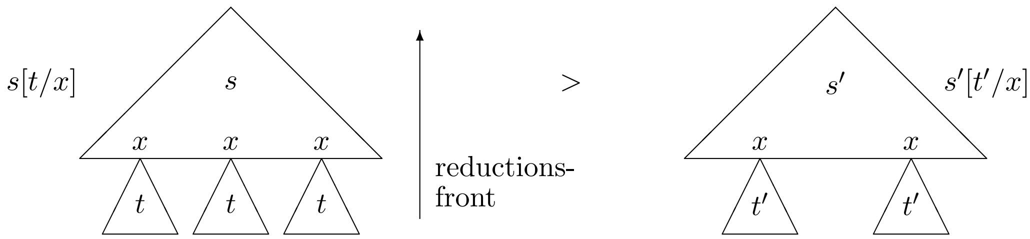
The next lemma follows directly from the analysis of applicable rules:

Lemma 1.2.12. $\lambda x.\, s > t \ \Rightarrow\ \exists s'.\ t = \lambda x.\, s' \ \wedge\ s > s'$

Lemma 1.2.13. $s > s' \ \wedge\ t > t' \ \Rightarrow\ s[t/x] > s'[t'/x]$

Proof:

By induction on s; in case $s = (s_1\ s_2)$, case distinction by applied rule is necessary. Details are left as exercises. The proof is graphically illustrated as follows:



Theorem 1.2.14. *$>$ has the diamond-property.*

Proof: we show $s > t_1 \ \wedge\ s > t_2 \ \Rightarrow\ \exists u.\, t_1 > u \ \wedge\ t_2 > u$ by induction on s.

1. s is an atom $\Rightarrow\ s = t_1 = t_2 =: u$

2. $s = \lambda x.\, s'$
   $\Rightarrow t_i = \lambda x.\, t_i'$ and $s' > t_i'$ (for $i = 1, 2$)
   $\Rightarrow \exists u'.\, t_i' > u' \quad (i = 1, 2)$ (by induction hypothesis)
   $\Rightarrow t_i = \lambda x.\, t_i' > \lambda x.\, u' =: u$

3. $s = (s_1\ s_2)$

   Case distinction by rules. Convention: $s_i > s_i', s_i''$ and $s_i', s_i'' > u_i$.

   (a) (By induction hypothesis)

$$\begin{array}{ccc} (s_1\ s_2) & >_3 & (s_1'\ s_2') \\ \vee_3 & & \vee_3 \\ (s_1''\ s_2'') & >_3 & (u_1\ u_2) \end{array}$$

   (b) (By induction hypothesis and Lemma 1.2.13)

$$\begin{array}{ccc} (\lambda x.\, s_1)s_2 & >_4 & s_1'[s_2'/x] \\ \vee_4 & & \vee \\ s_1''[s_2''/x] & > & u_1[u_2/x] \end{array}$$

   (c) (By induction hypothesis and Lemma 1.2.13)

$$\begin{array}{ccc} (\lambda x.\, s_1)s_2 & >_3 & (\lambda x.\, s_1')s_2' \\ \vee_4 & & \vee_4 \\ s_1''[s_2''/x] & > & u_1[u_2/x] \end{array}$$

From the Lemmas 1.2.10 and 1.2.11 and Theorem 1.2.14 with A.2.5, the following lemma is obtained directly

Corollary 1.2.15. *$\rightarrow_\beta$ is confluent.*

1.3 η-reduction

$$\lambda x.\,(t\;x) \;\to_\eta\; t \quad \text{if } x \notin FV(t)$$

Motivation for η-reduction: $\lambda x.\,(t\;x)$ and t behave identically as functions:

$$(\lambda x.\,(t\;x))u \to_\beta t\;u$$

if $x \notin FV(t)$.

Of course η-reduction is not allowed at the root only.

Definition 1.3.1. $C[\lambda x.\,(t\;x)] \;\to_\eta\; C[t] \quad$ if $x \notin FV(t)$.

Fact 1.3.2. $\to_\eta$ *terminates.*

We prove local confluence of $\to_\eta$. Confluence of $\to_\eta$ follows from local confluence because of termination and Newman's Lemma.

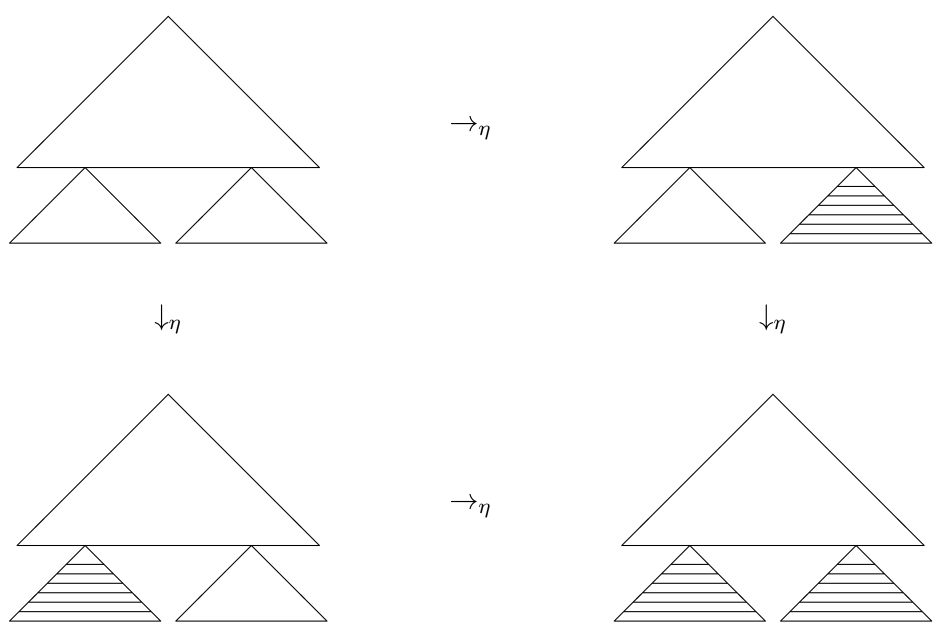
Fact 1.3.3. $s \to_\eta t \quad \Rightarrow \quad FV(s) = FV(t)$

Lemma 1.3.4. $\to_\eta$ *is locally confluent.*

$$\begin{tikzcd}
\bullet \arrow[r, "\eta"'] \arrow[d, "\eta"] & \bullet \arrow[d, dashed, "*"', "\eta"] \\
\bullet \arrow[r, dashed, "*", "\eta"'] & \bullet
\end{tikzcd}$$

Proof: by case discintion on the relative position of the two redexes in syntax tree of terms.

1. The redexes lie in separate subterms.



2. The positions of the redexes are identical. Obvious.
3. One redex is above the other. Proof by Fact 1.3.3.

$$\begin{array}{ccc} \lambda x.\, s\, x & \to_\eta & s \\ \downarrow_\eta & & \downarrow_\eta \\ \lambda x.\, s'\, x & \to_\eta & s' \end{array}$$

Corollary 1.3.5. $\to_\eta$ *is confluent.*

Proof: $\to_\eta$ terminates and is locally confluent.

Exercise: Define $\to_\eta$ inductively and prove the local confluence of $\to_\eta$ with help of that definition.

Remark:

> $\to_\eta$ does not have the diamond-property. But one can prove that $\overset{=}{\to}_\eta$ has the diamond-property by slightly modifiying Fact 1.3.3.

Lemma 1.3.6.

$$\begin{tikzcd} \bullet \arrow[r, "\beta"'] \arrow[d, "\eta"] & \bullet \arrow[d, dashed, "*\,\eta"'] \\ \bullet \arrow[r, dashed, "="', "\beta"'] & \bullet \end{tikzcd}$$

Proof: by case distinction on the relative position of redexes.

1. In separate subtrees: obvious
2. η-redex far below β-redex (no overlap):
   (a) $t \to_\eta t'$:

$$\begin{tikzcd} (\lambda x.\, s)t \arrow[r, "\beta"'] \arrow[d, "\eta"] & s[t/x] \arrow[d, dashed, "*\,\eta"'] \\ (\lambda x.\, s)t' \arrow[r, dashed, "\beta"'] & s[t'/x] \end{tikzcd}$$

   using the lemmas $t \to_\eta t' \Rightarrow s[t/x] \to_\eta^* s[t'/x]$.

   (b) $s \to_\eta s'$:

$$\begin{tikzcd} (\lambda x.\, s)t \arrow[r, "\beta"'] \arrow[d, "\eta"] & s[t/x] \arrow[d, dashed, "\eta"'] \\ (\lambda x.\, s')t \arrow[r, dashed, "\beta"'] & s'[t/x] \end{tikzcd}$$

3. β-redex ($s \to_\beta s'$) far below the η-redex (no overlap):

$$\begin{tikzcd} \lambda x.\, s\; x \arrow[r, "\beta"'] \arrow[d, "\eta"] & \lambda x.\, s'\; x \arrow[d, dashed, "\eta"] \\ s \arrow[r, dashed, "\beta"'] & s' \end{tikzcd}$$

   with help of exercise 1.2.8.

4. β-redex directly above the η-redex (they overlap):

$$\begin{tikzcd} (\lambda x.\, (s\; x))t \arrow[r, "\beta"'] \arrow[d, "\eta"] & s\; t \arrow[d, dashed, "*"', "\eta"] \\ s\; t \arrow[r, dashed, "="', "\beta"'] & s\; t \end{tikzcd}$$

5. β-redex directly below the η-redex (they overlap):

$$\begin{tikzcd} \lambda x.\, ((\lambda y.\, s)x) \arrow[r, "\beta"'] \arrow[d, "\eta"] & \lambda x.\, s[x/y] \arrow[d, dashed, "*"', "\eta"] \\ \lambda y.\, s \arrow[r, dashed, "=", "\beta"'] & \lambda y.\, s \end{tikzcd}$$

   because $\lambda y.\, s =_\alpha \lambda x.\, s[x/y]$ as $x \notin FV(s)$ due to $\lambda x.\, ((\lambda y.\, s)x) \to_\eta \lambda y.\, s$ □

By Lemma A.3.3, $\xrightarrow{*}_\beta$ and $\xrightarrow{*}_\eta$ commute. Since both are confluent, with the lemma of Hindley and Rosen the following corollary holds.

Corollary 1.3.7. $\to_{\beta\eta}$ *is confluent.*

1.4 λ-calculus as an equational theory

1.4.1 β-conversion

Definition 1.4.1 (equivalence modulo β-conversion)**.**

$$s =_\beta t \;:\Leftrightarrow\; s \leftrightarrow^*_\beta t$$

Alternatively:

$$(\lambda x.\, s)\; t =_\beta s[t/x] \qquad\qquad t =_\beta t$$

$$\frac{s =_\beta t}{\lambda x.\, s =_\beta \lambda x.\, t} \qquad \frac{s =_\beta t}{t =_\beta s} \qquad \frac{s_1 =_\beta t_1 \quad s_2 =_\beta t_2}{(s_1\; s_2) =_\beta (t_1\; t_2)} \qquad \frac{s =_\beta t \quad t =_\beta u}{s =_\beta u}$$

Since $\to_\beta$ is confluent, one can replace the test for equivalence with the search for a common reduction.

Theorem 1.4.2. *$s =_\beta t$ is decidable if s and t possess a β-normal form, otherwise undecidable.*

Proof: Decidability follows directly from Corollary A.2.8, since $\rightarrow_\beta$ is confluent. Undecidability follows from the fact that λ-terms are programs and program equivalences are undecidable. $\square$

1.4.2 η-conversion and extensionality

Extensionality means that two functions are equal if they are equal on all arguments:

$$\text{ext}: \quad \frac{\forall u. s\ u = t\ u}{s = t}$$

Theorem 1.4.3. *$\beta + \eta$ and $\beta + \text{ext}$ define the same equivalence on λ-terms.*

Proof:

$\eta \Rightarrow \text{ext}$: $\forall u. s\ u = t\ u \ \Rightarrow\ s\ x = t\ x$ where $x \notin FV(s,t) \ \Rightarrow\ s =_\eta \lambda x.\,(s\ x) = \lambda x.\,(t\ x) = t$

$\beta + \text{ext} \Rightarrow \eta$: let $x \notin FV(s)$: $\forall u. (\lambda x.\,(s\ x))u =_\beta s\ u \ \Rightarrow\ \lambda x.\,(s\ x) = s$ $\square$

Definition 1.4.4.

$$\begin{aligned} s \rightarrow_{\beta\eta} t \quad &:\Leftrightarrow \quad s \rightarrow_\beta t \ \vee\ s \rightarrow_\eta t \\ s =_{\beta\eta} t \quad &:\Leftrightarrow \quad s \leftrightarrow^*_{\beta\eta} t \end{aligned}$$

Analogously to $=_\beta$, we have the following theorem.

Theorem 1.4.5. *$s =_{\beta\eta} t$ is decidable if s and t possess a $\beta\eta$-normalform, otherwise undecidable.*

Since $\rightarrow_\eta$ is terminating and confluent, the following corollary holds.

Corollary 1.4.6. *$\leftrightarrow^*_\eta$ is decidable.*

1.5 Reduction strategies

The order in which β-redexes are contracted can influence if a normal form is reached or not. For example (where $(\Omega := (\lambda x.\,x\ x)(\lambda x.\,x\ x))$:

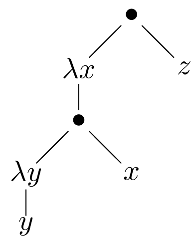
$$\begin{tikzcd} \overbrace{(\lambda x.\,5)(\Omega)} \arrow[loop right, "\text{call-by-value}"] \arrow[r, "\text{call-by-name}"] & 5 \end{tikzcd}$$

Theorem 1.5.1. *If t has a β-normal form, then this normal form can be reached by reducing the leftmost β-redex in each step. This is called **normal-order** reduction.*

A sequence of leftmost reductions where the λ in every β-redex is underlined:

$$\begin{array}{ll} & (\underline{\lambda} x.\,x\,(\lambda y.\,x\,y\,y)\,x)\,(\lambda z.\,\lambda w.\,z) \\ \rightarrow_\beta & (\underline{\lambda} z.\,\lambda w.\,z)\,(\lambda y.\,(\underline{\lambda} z.\,\lambda w.\,z)\,y\,y)\,(\lambda z.\,\lambda w.\,z) \\ \rightarrow_\beta & (\underline{\lambda} w.\,(\lambda y.\,(\underline{\lambda} z.\,\lambda w.\,z)\,y\,y)\,(\lambda z.\,\lambda w.\,z) \\ \rightarrow_\beta & (\lambda y.\,(\underline{\lambda} z.\,\lambda w.\,z)\,y\,y) \\ \rightarrow_\beta & (\lambda y.\,(\underline{\lambda} w.\,y)\,y) \\ \rightarrow_\beta & (\lambda y.\,y) \end{array}$$

The leftmost redex in the linear (string) representation of a term is the *leftmost outermost* (i.e. the leftmost of the outermost) β-redex in the tree representation. For example consider:



The leftmost redex in the string $(\lambda x.\, (\lambda y.\, y)\ x))\, z$ is not the leftmost redex in the tree but the leftmost of the outermost redexes.

Now for some precise inductive definitions.

Notation: $\overline{t_m}$ is a sequence of terms $t_1, \ldots, t_m$. If the number of terms is irrelevant we simply write $\overline{t}$.

The small-step **normal-order** reduction relation $\rightarrow_n$ reduces the leftmost outermost β-redex in each step.

Definition 1.5.2. The relation $\rightarrow_n$ is defined inductively:

$$\frac{t_1, \ldots, t_m \in NF \quad r \rightarrow_n r'}{x\,\overline{t_m}\,r\,\overline{s} \rightarrow_n x\,\overline{t_m}\,r'\,\overline{s}}\ (1) \qquad \frac{s \rightarrow_n t}{\lambda x.\, s \rightarrow_n \lambda x.\, t}\ (2) \qquad (\lambda x.\, r)\, s\, \overline{s} \rightarrow_n r[s/x]\,\overline{s}\ \ (3)$$

The relation $\rightarrow_n$ is deterministic: any term can reduce to at most one other term. This is not hard to see because each term can be reduced by at most one rule and each rule is deterministic. A proper proof requires induction.

The normal-order reduction relation $\Rightarrow_n$ defined below reduces a term in one big step. It is the big-step counterpart of the small-step relation $\rightarrow_n$ and can be viewed as inductive definition of a recursive normalization function.

Definition 1.5.3. The relation $\Rightarrow_n$ is defined inductively:

$$\frac{s_1 \Rightarrow_n t_1, \ldots, s_m \Rightarrow_n t_m}{x\,\overline{s_m} \Rightarrow_n x\,\overline{t_m}}\ (1) \qquad \frac{s \Rightarrow_n t}{\lambda x.\, s \Rightarrow_n \lambda x.\, t}\ (2) \qquad \frac{r[s/x]\,\overline{s} \Rightarrow_n t}{(\lambda x.\, r)\, s\, \overline{s} \Rightarrow_n t}\ (3)$$

Definition 1.5.4. The set NF of terms is defined inductively:

$$\frac{t_1, \ldots, t_m \in NF}{x\,\overline{t_m} \in NF}\ (1) \qquad \frac{t \in NF}{\lambda x.\, t \in NF}\ (2)$$

Lemma 1.5.5. *Term t is in β-normal form iff $t \in NF$.*

Lemma 1.5.6. *If $s \Rightarrow_n t$ then $t \in NF$.*

Proof. By induction on $s \Rightarrow_n t$.

Case (1): We have $s_i \Rightarrow_n t_i$ and $t_i \in NF$ (IH). Thus $x\,\overline{t_m} \in NF$ by (1).

Case (2): We have $s \Rightarrow_n t$ and $t \in NF$ (IH). Thus $\lambda x.\, t \in NF$ by (2).

Case (3): $t \in NF$ follows directly by IH. ☐

Theorem 1.5.7. *$s \Rightarrow_n t$ iff $s \xrightarrow{*}_n t$ and $t \in NF$.*

Theorem 1.5.8 (Standardization)**.** *If $s \xrightarrow{*}_\beta t$ and $t \in NF$ then there is a normal-order reduction sequence from s to t: $s \xrightarrow{*}_n t$.*

For a proof see, for example, Barendregt [Bar84].

1.5.1 Evaluation strategies in Programming Languages

Evaluation in programming languages is more restrictive than *reduction* in lambda calculus: terms must be closed and there is no reduction under λs. More precisely, evaluation stops as soon as a *value* has been reached. In our simple setting, the only values are λ-abstractions:

$$v \ ::= \ \lambda x.\, t$$

Call-by-name is a restriction of normal-order reduction. This is a small-step formulation:

Definition 1.5.9. The relation $\to_{cbn}$ is defined inductively:

$$(\lambda x.\, r)\, s \to_{cbn} r[s/x] \quad (1) \qquad \frac{r \to_{cbn} r'}{r\, s \to_{cbn} r'\, s}\ (2)$$

Call-by-name reduction is deterministic.

In contrast to call-by-name, **call-by-value** evaluates the arguments before substituting them into the function. This is a small-step formulation:

Definition 1.5.10. The relation $\to_{cbv}$ is defined inductively:

$$(\lambda x.\, r)\, v \to_{cbv} r[v/x] \quad (1) \qquad \frac{r \to_{cbv} r'}{r\, s \to_{cbv} r'\, s}\ (2) \qquad \frac{r \to_{cbv} r'}{v\, r \to_{cbv} v\, r'}\ (3)$$

Call-by-value reduction is also deterministic.

1.6 Labeled terms

Motivation: `let`-expression

$$\texttt{let}\ x = s\ \texttt{in}\ t \ \to_{\texttt{let}}\ t[s/x]$$

`let` can be interpreted as labeled β-redex. Example:

$$\begin{tikzcd}
\texttt{let}\ x = (\texttt{let}\ y = s\ \texttt{in}\ y + y)\ \texttt{in}\ x * x \arrow[r] \arrow[d] & \texttt{let}\ x = s + s\ \texttt{in}\ x * x \arrow[d, dashed] \\
(\texttt{let}\ y = s\ \texttt{in}\ y + y) * (\texttt{let}\ y = s\ \texttt{in}\ y + y) \arrow[r, dashed] & (s + s) * (s + s)
\end{tikzcd}$$

Set of labeled terms $\underline{\mathcal{T}}$ is defined as follows:

$$t \quad ::= \quad c \quad | \quad x \quad | \quad (t_1\ t_2) \quad | \quad \lambda x.t \quad | \quad (\underline{\lambda} x.s)\ t$$

Note: $\underline{\lambda} x.s \notin \underline{\mathcal{T}}$ (why?)

Definition 1.6.1. $\underline{\beta}$-reduction of labeled terms:

$$C[(\underline{\lambda} x.s)\ t] \to_{\underline{\beta}} C[s[t/x]]$$

Goal: $\to_{\underline{\beta}}$ terminates.

Property: $\to_{\underline{\beta}}$ cannot generate new labeled redexes, but can only copy and modify existing redexes. The following example shall illustrate the difference between $\to_\beta$ and $\to_{\underline{\beta}}$:

$$(\lambda x.x\ x)(\lambda x.x\ x) \to_\beta \underbrace{(\lambda x.x\ x)(\lambda x.x\ x)}_{\text{new } \beta\text{-redex}}$$

but

$$(\underline{\lambda} x.x\ x)(\lambda x.x\ x) \to_{\underline{\beta}} \underbrace{(\lambda x.x\ x)(\lambda x.x\ x)}_{\text{no } \underline{\beta}\text{-redex}}$$

If $s \to_{\underline{\beta}} t$, then every $\underline{\beta}$-redex in t derives from exactly one $\underline{\beta}$-redex in s.

In the following, let $s[t_1/x_1, \ldots, t_n/x_n]$ be the simultaneous substitution of x_i by t_i in s.

Lemma 1.6.2.

1. $s, t_1, \ldots, t_n \in \underline{\mathcal{T}} \;\Rightarrow\; s[t_1/x_1, \ldots, t_n/x_n] \in \underline{\mathcal{T}}$
2. $s \in \underline{\mathcal{T}} \;\wedge\; s \to_{\underline{\beta}} t \;\Rightarrow\; t \in \underline{\mathcal{T}}$

Exercise 1.6.3. Prove this lemma.

Theorem 1.6.4. *Let $s, t_1, \ldots, t_n \in \underline{\mathcal{T}}$. Then $s[t_1/x_1, \ldots, t_n/x_n]$ terminates with regard to $\to_{\underline{\beta}}$ if every t_i terminates.*

Proof: by induction on s. Set $[\sigma] := [t_1/x_1, \ldots, t_n/x_n]$.

1. s is a constant: obvious
2. s is a variable:
   - $\forall i.\, s \neq x_i$: obvious
   - $s = x_i$: obvious since t_i terminates
3. $s = (s_1\ s_2)$:
   $s[\sigma] = (s_1[\sigma])(s_2[\sigma])$ terminates, because $s_i[\sigma]$ terminates (Ind.-Hyp.), and $s_1[\sigma] \to^*_{\underline{\beta}} \underline{\lambda} x.t$ is impossible due to Lemma 1.6.2, since $s_1[\sigma] \in \underline{\mathcal{T}}$ but $\underline{\lambda} x.t \notin \underline{\mathcal{T}}$.
4. $s = \lambda x.t$: $s[\sigma] = \lambda x.(t[\sigma])$ terminates since $t[\sigma]$ terminates (Ind.-Hyp.).
5. $s = (\underline{\lambda} x.t)u$:
   $s[\sigma] = (\underline{\lambda} x.(t[\sigma]))(u[\sigma])$, where $t[\sigma]$ and $u[\sigma]$ terminate (Ind.-Hyp.). Every infinite reduction would look like this:

   $$s[\sigma] \quad \to^*_{\underline{\beta}} \quad (\underline{\lambda} x.t')\ u' \quad \to_{\underline{\beta}} \quad t'[u'/x] \quad \to_{\underline{\beta}} \quad \ldots$$

   But: Since $u[\sigma]$ terminates and $u[\sigma] \to^*_{\underline{\beta}} u'$, u' must also terminate. Since $t[\sigma] \to^*_{\underline{\beta}} t'$, the following also holds:

   $$\underbrace{t[\sigma, u'/x]}_{\substack{\text{This terminates by Ind.-Hyp.,}\\ \text{since } \sigma \text{ and } u' \text{ terminate.}}} \qquad \to^*_{\underline{\beta}} \qquad \underbrace{t'[u'/x]}_{\substack{\text{So, this must also}\\ \text{terminate.}}}$$

   $\Rightarrow$ Contradiction to the assumption that there is an infinite reduction. □

Corollary 1.6.5. $\to_{\underline{\beta}}$ *terminates for all terms in $\underline{\mathcal{T}}$.*

Length of reduction sequence: not more than exponential in the size of the input term.

Theorem 1.6.6. *$\rightarrow_{\underline{\beta}}$ is confluent.*

Proof: $\rightarrow_{\underline{\beta}}$ is locally confluent. (Use termination and Newman's Lemma.) □

Connection between $\rightarrow_{\underline{\beta}}$ and the parallel reduction $>$:

Theorem 1.6.7. *Let $|\underline{s}|$ the unlabeled version of $\underline{s} \in \underline{\mathcal{T}}$. Then,*

$$s > t \quad \Leftrightarrow \quad \exists \underline{s} \in \underline{\mathcal{T}}.\ \underline{s} \rightarrow^*_{\underline{\beta}} t \wedge |\underline{s}| = s$$

1.7 Lambda calculus as a programming language

1.7.1 Data types

- `bool`:

  $$\begin{array}{lll} \texttt{true}, \texttt{false}, \texttt{if} & \text{with} & \texttt{if}\ \texttt{true}\ x\ y \rightarrow^*_\beta x \\ & \text{and} & \texttt{if}\ \texttt{false}\ x\ y \rightarrow^*_\beta y \end{array}$$

  is realized by

  $$\begin{array}{rcl} \texttt{true} & = & \lambda xy.x \\ \texttt{false} & = & \lambda xy.y \\ \texttt{if} & = & \lambda zxy.z\ x\ y \end{array}$$

- Pairs:

  $$\begin{array}{lll} \texttt{fst}, \texttt{snd}, \texttt{pair} & \text{with} & \texttt{fst}(\texttt{pair}\ x\ y) \rightarrow^*_\beta x \\ & \text{and} & \texttt{snd}(\texttt{pair}\ x\ y) \rightarrow^*_\beta y \end{array}$$

  is realized by

  $$\begin{array}{rcl} \texttt{fst} & = & \lambda p.p\ \texttt{true} \\ \texttt{snd} & = & \lambda p.p\ \texttt{false} \\ \texttt{pair} & = & \lambda xy.\lambda z.z\ x\ y \end{array}$$

  Example:

  $$\begin{array}{lllllll} \texttt{fst}(\texttt{pair}\ x\ y) & \rightarrow_\beta & \texttt{fst}(\lambda z.z\ x\ y) & \rightarrow_\beta & (\lambda z.z\ x\ y)(\lambda xy.x) & & \\ & \rightarrow_\beta & (\lambda x\ y.x)\ x\ y & \rightarrow_\beta & (\lambda y.x)\ y & \rightarrow_\beta & x \end{array}$$

- `nat` (Church-Numerals):

  $$\begin{array}{rcl} \underline{0} & = & \lambda f.\lambda x.x \\ \underline{1} & = & \lambda f.\lambda x.f\ x \\ \underline{2} & = & \lambda f.\lambda x.f(f\ x) \\ & \vdots & \\ \underline{n} & = & \lambda f.\lambda x.f^n(x) = \lambda f.\lambda x.\underbrace{f(f(\ldots f}_{n\text{-times}}(x)\ldots)) \end{array}$$

Arithmetic:

$$\begin{aligned}\texttt{succ} &= \lambda n.\lambda f\,x.f(n\ f\ x)\\ \texttt{add} &= \lambda m\,n.\lambda f\,x.m\ f(n\ f\ x)\\ \texttt{iszero} &= \lambda n.n(\lambda x.\texttt{false})\ \texttt{true}\end{aligned}$$

Therefore:

$$\begin{array}{llllll}\texttt{add}\ \underline{n}\ \underline{m} & \to^2 & \lambda f\,x.\underline{n}\ f(\underline{m}\ f\ x) & \to^2 & \lambda f\,x.\underline{n}\ f(f^m(x)) & \\ & \to^2 & \lambda f\,x.f^n(f^m(x)) & = & \lambda f\,x.f^{n+m}(x) & = \quad \underline{n+m}\end{array}$$

Exercise 1.7.1.

1. Lists in λ-calculus: Find λ-terms for `nil`, `cons`, `hd`, `tl`, `null` with

$$\begin{array}{ll}\texttt{null nil} \to^* \texttt{true} & \texttt{hd}(\texttt{cons}\ x\ l) \to^* x\\ \texttt{null}(\texttt{cons}\ x\ l) \to^* \texttt{false} & \texttt{tl}(\texttt{cons}\ x\ l) \to^* l\end{array}$$

   Hint: Use Pairs.

2. Find `mult` with $\texttt{mult}\ \underline{m}\ \underline{n} \overset{*}{\to} \underline{m*n}$
   and `expt` with $\texttt{expt}\ \underline{m}\ \underline{n} \overset{*}{\to} \underline{m^n}$

3. Difficult: Find `pred` with $\texttt{pred}\ \underline{m+1} \overset{*}{\to} \underline{m}$ and $\texttt{pred}\ \underline{0} \overset{*}{\to} \underline{0}$

1.7.2 Recursive functions

Given a recursive function $f(x) = e$, we look for a non-recursive representation $f = t$. Note: $f(x) = e$ is not a definition in the mathematical sense, but only a (not uniquely) characterizing property.

$$\begin{array}{ll} & f(x) = e\\ \Rightarrow & f = \lambda x.e\\ \Rightarrow & f =_\beta (\lambda f.\lambda x.e)\ f\\ \Rightarrow & f \text{ is fixed point of } F := \lambda f x.e, \text{ i.e. } f =_\beta F\ f\end{array}$$

Let `fix` be a fixed point operator, i.e. $\texttt{fix}\ t =_\beta t(\texttt{fix}\ t)$ for all terms t. Then f can be defined non-recursively as follows

$$f := \texttt{fix}\ F$$

Recursive f and non-recursive f behave identically:

1. recursive:

$$f\ s = (\lambda x.e)\ s \to_\beta e[s/x]$$

2. non-recursive:

$$f\ s\ =\ \texttt{fix}\ F\ s\ =_\beta\ F\ (\texttt{fix}\ F)\ s\ =\ F\ f\ s\ \to_\beta^2\ e[f/f, s/x]\ =\ e[s/x]$$

Example:

$$\texttt{add}\ m\ n = \texttt{if}\ (\texttt{iszero}\ m)\ n\ (\texttt{add}\ (\texttt{pred}\ m)\ (\texttt{succ}\ n))$$

$$\texttt{add} := \texttt{fix}\ \underbrace{(\lambda\texttt{add}.\lambda m\,n.\texttt{if}\ (\texttt{iszero}\ m)\ n\ (\texttt{add}\ (\texttt{pred}\ m)(\texttt{succ}\ n)))}_{A}$$

$$
\begin{array}{rcl}
\texttt{add}\ \underline{1}\ \underline{2} & = & \texttt{fix}\ A\ \underline{1}\ \underline{2} \\
& =_\beta & A\ (\texttt{fix}\ A)\ \underline{1}\ \underline{2} \\
& \rightarrow_\beta^3 & \texttt{if}\ (\texttt{iszero}\ \underline{1})\ \underline{2}\ (\texttt{fix}\ A\ (\texttt{pred}\ \underline{1})\ (\texttt{succ}\ \underline{2})) \\
& \rightarrow_\beta^* & \texttt{fix}\ A\ \underline{0}\ \underline{3} \\
& =_\beta & A\ (\texttt{fix}\ A)\ \underline{0}\ \underline{3} \\
& \rightarrow_\beta^3 & \texttt{if}\ (\texttt{iszero}\ \underline{0})\ \underline{3}\ (...) \\
& \rightarrow_\beta^* & \underline{3}
\end{array}
$$

Note: even $\texttt{add}\ \underline{1}\ \underline{2} \overset{*}{\rightarrow}_\beta \underline{3}$ holds. Why?

We now show that `fix`, i.e. the fixed point operator, can be defined in pure λ-calculus. The two most well-known solutions are:

Church: $V_f := \lambda x.f(x\ x)$ and $Y := \lambda f.V_f\ V_f$

Y is called "Church's fixed-point combinator"

$$Y\ t\ \rightarrow_\beta\ V_t\ V_t\ \rightarrow_\beta\ t(V_t\ V_t)\ \leftarrow_\beta\ t((\lambda f.V_f\ V_f)t)\ =\ t(Y\ t)$$

Therefore: $Y\ t =_\beta t(Y\ t)$

Turing: $A := \lambda x\,f.f(x\ x\ f)$ and $\Theta := A\ A \rightarrow_\beta \lambda f.f(A\ A\ f)$. Therefore

$$\Theta\ t\ =\ A\ A\ t \rightarrow_\beta (\lambda f.f(A\ A\ f))t\ \rightarrow_\beta\ t(A\ A\ t)\ =\ t(\Theta\ t)$$

Therefore: $\Theta\ t \rightarrow_\beta^* t(\Theta\ t)$

1.7.3 Computable functions on $\mathbb{N}$

Definition 1.7.2. A (possibly partial) function $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is **λ-definable** if there exists a closed pure λ-term (without free variables!) with

1. $t\ \underline{m_1}\ \ldots\ \underline{m_n} \rightarrow_\beta^* \underline{m}$, if $f(m_1, \ldots, m_n) = m$
2. $t\ \underline{m_1}\ \ldots\ \underline{m_n}$ has no β-normal form, if $f(m_1, \ldots, m_n)$ is undefinied.

Theorem 1.7.3. *All the Turing machine-computable functions (*`while`*-computable, μ-recursive) are lambda-definable, and vice versa.*

Chapter 2

Combinatory logic (CL)

Keyword: "variable-free programming"

Terms:

$$X \quad ::= \quad \underbrace{x}_{\text{variables}} \quad | \quad \underbrace{\mathsf{S} \ | \ \mathsf{K} \ | \ \mathsf{I} \ | \ \ldots}_{\text{constants}} \quad | \quad X_1\, X_2 \quad | \quad (X)$$

Application associates to the left as usual: $\quad X\ Y\ Z \quad = \quad (X\ Y)\ Z$

Combinators are variable-free terms. (More precisely: they contain only S and K.)

Calculation rules for **weak reduction** (**weak reduction**, $\rightarrow_{\mathrm{w}}$):

$$\begin{aligned}
\mathsf{I}\, X &\rightarrow_{\mathrm{w}} X \\
\mathsf{K}\, X\, Y &\rightarrow_{\mathrm{w}} X \\
\mathsf{S}\, X\ Y\ Z &\rightarrow_{\mathrm{w}} (X\ Z)(Y\ Z) \\
X \rightarrow_{\mathrm{w}} X' \quad \Rightarrow \quad & X\ Y \rightarrow_{\mathrm{w}} X'\, Y \quad \wedge \quad Y\ X \rightarrow_{\mathrm{w}} Y\ X'
\end{aligned}$$

Examples:

1. $\mathsf{S}\,\mathsf{K}\,X\,Y \quad \rightarrow_{\mathrm{w}} \quad \mathsf{K}\,Y\,(X\,Y) \quad \rightarrow_{\mathrm{w}} \quad Y$
2. $\mathsf{S}\,\mathsf{K}\,\mathsf{K}\,X \quad \rightarrow_{\mathrm{w}} \quad \mathsf{K}\,X\,(\mathsf{K}\,X) \quad \rightarrow_{\mathrm{w}} \quad X$

We see that $\mathsf{S}\ \mathsf{K}\ \mathsf{K}$ and I behave identically. Therefore I is theoretically unnecessary, but it is useful in practice.

Theorem 2.0.1. $\rightarrow_{\mathrm{w}}$ *is confluent.*

Proof possibilities:

1. Proof by parallel reduction. This is simpler than the proof for $\rightarrow_\beta$.
2. Proof by "Each orthogonal term rewriting system is confluent."

The term rewriting system $\rightarrow_{\mathrm{w}}$ is not terminating:

Exercise 2.0.2. Find a comtinator X with $X \rightarrow_{\mathrm{w}}^{+} X$.

Exercise 2.0.3. Find combinators A, W, B with

$$\begin{array}{rcl} \mathsf{A}\,X & \rightarrow_{\mathrm{w}}^{*} & X\,X \\ \mathsf{W}\,X\,Y & \rightarrow_{\mathrm{w}}^{*} & X\,Y\,Y \\ \mathsf{B}\,X\,Y\,Z & \rightarrow_{\mathrm{w}}^{*} & X\,(Y\,Z) \end{array}$$

Theorem 2.0.4. *If a CL-term has a normal form, then one can find this normal form by always reducing the leftmost-outermost $\rightarrow_{\mathrm{w}}$-redex.*

2.1 Relationship between λ-calculus and CL

Translation of λ-terms into CL-terms:

$$(\_)_{\mathrm{CL}}: \qquad \lambda\text{-Terme} \quad \rightarrow \quad \text{CL-Terme}$$

$$\begin{array}{rcl} (x)_{\mathrm{CL}} & = & x \\ (s\ t)_{\mathrm{CL}} & = & (s)_{\mathrm{CL}}\ (t)_{\mathrm{CL}} \\ (\lambda x.s)_{\mathrm{CL}} & = & \lambda^{*}x.(s)_{\mathrm{CL}} \end{array}$$

Auxiliary function λ^{*}: Vars $\times$ CL-terms $\rightarrow$ CL-terms

$$\begin{array}{rccl} \lambda^{*}x.x & = & \mathsf{I} & \\ \lambda^{*}x.X & = & \mathsf{K}\ X & \text{if } x \notin FV(X) \\ \lambda^{*}x.(X\ Y) & = & \mathsf{S}(\lambda^{*}x.X)(\lambda^{*}x.Y) & \text{if } x \in FV(X\ Y) \end{array}$$

Lemma 2.1.1. $(\lambda^{*}x.X)\ Y \quad \rightarrow_{\mathrm{w}}^{*} \quad X[Y/x]$

Proof: by structural induction on X

- if $X \equiv x$: $(\lambda^{*}x.X)Y = \mathsf{I}\ Y \rightarrow_{\mathrm{w}} Y = X[Y/x]$
- if x in X is not free: $(\lambda^{*}x.X)Y = \mathsf{K}\ X\ Y \rightarrow_{\mathrm{w}} X = X[Y/x]$
- if $X \equiv U\ V$ and $x \in FV(X)$:

$$\begin{array}{rcccc} (\lambda^{*}x.(U\ V))Y & = & \mathsf{S}(\lambda^{*}x.U)(\lambda^{*}x.V)Y & \rightarrow_{\mathrm{w}} & ((\lambda^{*}x.U)Y)((\lambda^{*}x.V)Y) \\ (\text{Ind.-Hyp.}) & \rightarrow_{\mathrm{w}} & (U[Y/x])(V[Y/x]) & = & X[Y/x] \end{array}$$

Translation of CL-terms into λ-terms:

$$(\_)_{\lambda}: \qquad \text{CL-Terme} \quad \rightarrow \quad \lambda\text{-Terme}$$

$$\begin{array}{rcl} (x)_{\lambda} & = & x \\ (\mathsf{K})_{\lambda} & = & \lambda xy.x \\ (\mathsf{S})_{\lambda} & = & \lambda xyz.x\ z\ (y\ z) \\ (X\ Y)_{\lambda} & = & (X)_{\lambda}\ (Y)_{\lambda} \end{array}$$

Theorem 2.1.2. $((s)_{\mathrm{CL}})_{\lambda} \quad \rightarrow_{\beta}^{*} \quad s$

Proof: by structural induction on s:

1. $((a)_{\mathrm{CL}})_\lambda = a$
2. By Ind.-Hyp.: $((t\ u)_{\mathrm{CL}})_\lambda = ((t)_{\mathrm{CL}}\ (u)_{\mathrm{CL}})_\lambda = ((t)_{\mathrm{CL}})_\lambda((u)_{\mathrm{CL}})_\lambda \overset{*}{\to}_\beta t\ u$
3. By lemma 2.1.3 and Ind.-Hyp.: $((\lambda x.t)_{\mathrm{CL}})_\lambda = (\lambda^* x.(t)_{\mathrm{CL}})_\lambda \overset{*}{\to}_\beta \lambda x.((t)_{\mathrm{CL}})_\lambda \overset{*}{\to}_\beta \lambda x.t$ $\square$

Lemma 2.1.3. $(\lambda^* x.P)_\lambda \overset{*}{\to}_\beta \lambda x.(P)_\lambda$

Proof: exercise.

Corollary 2.1.4. S *and* K *are sufficient to represent all the* λ*-terms:* $\forall s \exists X.\, (X)_\lambda =_\beta s$

Proof: set $X := (s)_{\mathrm{CL}}$

Exercise 2.1.5. Show that B, C, K and W are also sufficient to represent all λ-terms () (Here: $\mathsf{C}\,X\,Y\,Z \ \to_{\mathrm{w}}\ X\,Z\,Y$). Is it possible to leave out K as well?

Theorem 2.1.6. $((X)_\lambda)_{\mathrm{CL}} =_{\mathrm{w,ext}} X$ *where* $=_{\mathrm{w}}\ :=\ \leftrightarrow^*_{\mathrm{w}}$ *and*

$$\text{(ext)}: \quad \frac{\forall x. X\ x =_{\mathrm{w,ext}} Y\ x}{X =_{\mathrm{w,ext}} Y} \qquad \text{(extensionality)}$$

Theorem 2.1.7. $X \to_{\mathrm{w}} Y \quad \Rightarrow \quad (X)_\lambda \to^*_\beta (Y)_\lambda$

Proof.

$$\begin{tikzcd} C[\mathsf{K}\ X\ Y] \arrow[r, "\mathrm{w}"'] \arrow[d, mapsto, "\lambda"] & C[X] \arrow[d, mapsto, "\lambda"] \\ C_\lambda[(\lambda xy.x)\ X_\lambda\ Y_\lambda] \arrow[r, "*", "\beta"'] & C_\lambda[X_\lambda] \end{tikzcd}$$

Similarly for S. $\square$

But: in general $s \to_\beta t$ does *not* imply $(s)_{\mathrm{CL}} \to^*_{\mathrm{w}} (t)_{\mathrm{CL}}$. Exercise: Find a counterexample!

Corollary 2.1.8. *If* $(t)_{\mathrm{CL}} \to^*_{\mathrm{w}} X$ *then* $t =_\beta (X)_\lambda$ *because* $t\ {}^*{}_\beta\!\leftarrow ((t)_{\mathrm{CL}})_\lambda \to^*_\beta (X)_\lambda$ *by Theorems 2.1.2 and 2.1.7.*

2.2 Implementation issues

Problems with the effective implementation of $\to_\beta$:

- Naive implementation by copying is very inefficient!
- Copying is sometimes necessary.

  Example: Let $t := \lambda x.(f\ x)$.

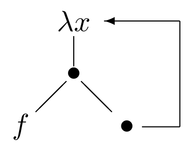
$$\begin{tikzcd}[column sep=small] (\lambda x.(x\ x))t \arrow[r, phantom, "\to_\beta"] & (\bullet \arrow[dr] & & \bullet) \arrow[dl] \\ & & t & \end{tikzcd}$$

with a copy:

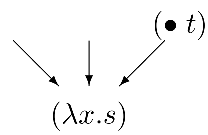
$$\dots \quad \to_\beta \quad f(\lambda x.f\ x)$$

Without a copy, a cyclic term arises:

$$\dots \quad \to_\beta$$



generally:



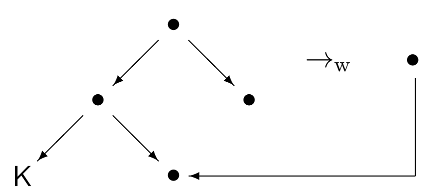
For β-reduction of $(\bullet\ t)$ copy of s is necessary!

- α-conversion is necessary.

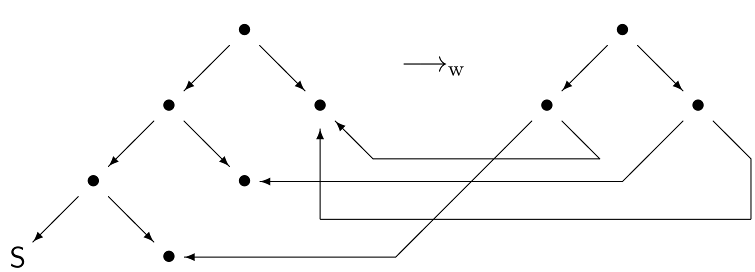
Graph reduction

A radical solution is the translation into CL, because $\to_{\mathrm{w}}$ is implemented on graphs without copying:

1. $(\mathsf{K}\ x)\ y \quad \to_{\mathrm{w}} \quad x$:



2. $\mathsf{S}\ x\ y\ z \quad \to_{\mathrm{w}} \quad x\ z\ (y\ z)$:



Here the problem is that $(\cdot)_{\mathrm{CL}}$ terms can get fairly large. But this problem can be compensated by optimization (replace S and K by optimal combinators). However, the structure of λ-terms always gets lost.

De Bruijn Notation

A second solution is the so-called de Bruijn indices:

$$\lambda x.\, \lambda y.\, (x\ z) \quad \cong \quad \lambda\ \lambda\ (1\ 2)$$

Bound variables are indices that indicate how many λs one must go through to get to the binding site. The syntax is therefore

$$t \quad ::= \quad i \ \mid\ \lambda t \ \mid\ (t_1\ t_2)$$

Examples:

$$\begin{array}{rcl} \lambda x.x & \cong & \lambda 0 \\ \lambda x.(y\ z) & \cong & \lambda(1\ 2) \end{array}$$

De Bruijn terms are difficult to read because the same bound variable can appear with different indexes. Example:

$$\lambda x.x\ (\lambda y.y\ x) \quad \cong \quad \lambda(0(\lambda(0\ 1)))$$

But: α-equivalent terms are identical in this notation!

We now consider β-reduction and substitution. Examples:

$$\begin{array}{rcl} \lambda x.(\lambda y.\lambda z.y)x & \to_\beta & \lambda x.\lambda z.x \\ \lambda((\lambda\lambda 1)0) & \to_\beta & \lambda\lambda 1 \end{array}$$

In general:

$$(\lambda s)t \quad \to_\beta \quad s[t/0]$$

where $s[t/i]$ means replacing i in s by t, where free variables in t may need to be incremented, and decrementing all free variables $\geq i$ in s by 1. Formally:

$$\begin{array}{rcl} j[t/i] & = & \texttt{if } i = j \texttt{ then } t \texttt{ else if } j > i \texttt{ then } j-1 \texttt{ else } j \\ (s_1\ s_2)[t/i] & = & (s_1[t/i])(s_2[t/i]) \\ (\lambda s)[t/i] & = & \lambda(s[\texttt{lift}(t,0)/i+1]) \end{array}$$

where $\texttt{lift}(t,i)$ means incrementing all variables $\geq i$ in t by 1. Formally:

$$\begin{array}{rcl} \texttt{lift}(j,i) & = & \texttt{if}'\ j \geq i \texttt{ then } j+1 \texttt{ else } j \\ \texttt{lift}((s_1\ s_2),i) & = & (\texttt{lift}(s_1,i))(\texttt{lift}(s_2,i)) \\ \texttt{lift}(\lambda s,i) & = & \lambda(\texttt{lift}(s,i+1)) \end{array}$$

Example:

$$\begin{array}{rcccccccc} (\lambda xy.x)\ z & \cong & (\lambda\lambda 1)0 & \to_\beta & (\lambda 1)[0/0] & = & \lambda(1[\texttt{lift}(0,0)/1]) & = \\ & = & \lambda\ (1[1/1]) & = & \lambda\ 1 & \cong & \lambda y.z & \end{array}$$

Chapter 3

Typed Lambda Calculi

Why types ?

1. To avoid inconsistency.

   Gottlob Frege's predicate logic ($\approx$ 1879) allows unlimited quantification over predicate.

   Russel (1901) discovers the paradox $\{X \mid X \notin X\}$.

   Whitehead & Russel's *Principia Mathematica* (1910–1913) forbids $X \in X$ using a type system based on "levels".

   Church (1930) invents the untyped λ-calculus as a logic.

   $\texttt{True}, \texttt{False}, \wedge, \ldots$ are λ-terms

   $\{x \mid P\} \equiv \lambda x.P \qquad x \in M \equiv Mx$

   inconsistence: $\quad R := \lambda x.\, \texttt{not}\,(x\ x) \quad \Rightarrow \quad R\, R =_\beta \texttt{not}\,(R\ R)$

   Church's simply typed λ-calculus (1940) forbids $x\ x$ with a type system.

2. To avoid programming errors.
   Classification of type systems:

   monomorphic: Each identifier has exactly one type.

   polymorphic: An identifier can have multiple types.

   static: Type correctness is checked at compile time.

   dynamic: Type correctness is checked at run time.

	static	dynamic
monomorphic	Pascal	
polymorphic	ML, Haskell (C++,) Java	Lisp, Smalltalk

3. To express specifications as types.

   Method: dependent types

   Example: mod: nat $\times$ m:nat $\rightarrow \{k \mid 0 \leq k < m\}$

   Result type depends on the input value

   This approach is known as "type theory".

3.1 Simply typed λ-calculus ($\lambda^{\rightarrow}$)

The simply typed λ-calculus is the heart of any typed (functional) programming language. Its types are built up from base types via the function space constructor according to the following grammar, where τ always represents a type:

$$\tau \quad ::= \quad \underbrace{\texttt{bool} \mid \texttt{nat} \mid \texttt{int} \mid \ldots}_{\text{basic types}} \mid \tau_1 \rightarrow \tau_2 \mid (\tau)$$

Convention: $\rightarrow$ associates to the right:

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau_3 \quad \equiv \quad \tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$$

Terms:

1. implicitly typed: terms as in the pure untyped λ-calculus, but each variable has a unique (implicit) type.
2. explicitly typed terms: $\quad t \ ::= \ x \ \mid \ (t_1 \ t_2) \ \mid \ \lambda x : \tau.t$

In both cases these are so-called "raw" typed terms, which are not necessarily type-correct, e.g. $\lambda x : int.(x \ x)$.

3.1.1 Type checking for explicitly typed terms

The goal is the derivation of statements of the form $\Gamma \vdash t : \tau$, i.e. t has the type τ in the context Γ. Here Γ has a finite function from variables to types. Notation: $[x_1 : \tau_1, \ldots, x_n : \tau_n]$. The notation $\Gamma[x : \tau]$ means to override Γ by the mapping $x \mapsto \tau$. Formally:

$$(\Gamma[x : \tau])(y) = \begin{cases} \tau & \text{if } x = y \\ \Gamma(y) & \text{otherwise} \end{cases}$$

Type checking rules:

$$\frac{\Gamma(x) \text{ is defined}}{\Gamma \vdash x : \Gamma(x)} \ (\text{Var})$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (t_1 \ t_2) : \tau_2} \ (\text{App}) \qquad\qquad \frac{\Gamma[x : \tau] \vdash t : \tau'}{\Gamma \vdash \lambda x : \tau.t : \tau \rightarrow \tau'} \ (\text{Abs})$$

Examples:

- A simple derivation:

$$\frac{\Gamma[x : \tau] \vdash x : \tau}{\Gamma \vdash \lambda x : \tau.x : \tau \rightarrow \tau}$$

- Not every term has a type. There are no context Γ and types τ and τ' such that $\Gamma \vdash \lambda x : \tau.(x \ x) : \tau'$, because

$$\cfrac{\cfrac{\cfrac{\tau = \tau_2 \rightarrow \tau_1}{\Gamma[x : \tau] \vdash x : \tau_2 \rightarrow \tau_1} \quad \cfrac{\tau = \tau_2}{\Gamma[x : \tau] \vdash x : \tau_2}}{\Gamma[x : \tau] \vdash (x \ x) : \tau_1} \qquad \tau' = \tau \rightarrow \tau_1}{\Gamma \vdash \lambda x : \tau.(x \ x) : \tau'}$$

$\Rightarrow$ Contradiction: $\neg \exists \tau_1, \tau_2 : \ \tau_2 \rightarrow \tau_1 = \tau_2$

The type checking rules constitute an algorithm for type checking by applying them backwards as in Prolog. In a functional style this becomes a function *type* that takes a context and a term and computes the type of the term or fails:

$$
\begin{array}{lcl}
\mathit{type}\ \Gamma\ x & = & \Gamma(x) \\
\mathit{type}\ \Gamma\ (t_1\ t_2) & = & \texttt{let}\ \tau_1 = \mathit{type}\ \Gamma\ t_1 \\
 & & \qquad \tau_2 = \mathit{type}\ \Gamma\ t_2 \\
 & & \texttt{in case}\ \tau_1\ \texttt{of} \\
 & & \qquad\quad \tau \to \tau' \Rightarrow \texttt{if}\ \tau = \tau_2\ \texttt{then}\ \tau'\ \texttt{else fail} \\
 & & \qquad \mid\ \ \_ \qquad\ \ \Rightarrow \texttt{fail} \\
\mathit{type}\ \Gamma\ (\lambda x : \tau.t) & = & \tau \to \mathit{type}\ (\Gamma[x : \tau])\ t
\end{array}
$$

Definition 3.1.1. t is **type-correct** (with regard to Γ), if there exists τ such that $\Gamma \vdash t : \tau$.

Lemma 3.1.2. *The type of a type-correct term is uniquely determined (with respect to a fixed context Γ).*

This follows because there is exactly one rule for each syntactic form of term: the rules are *syntax-directed*. Hence we are dealing with a monomorphic type system.

Lemma 3.1.3. *Each subterm of a type-correct term is type-correct.*

This is obvious from the rules.

Typed terms are closed under substitution:

Lemma 3.1.4. *If $\Gamma[x : \tau] \vdash s : \tau'$ and $\Gamma \vdash t : \tau$ then $\Gamma \vdash s[t/x] : \tau'$.*

The proof is by induction on $\Gamma[x : \tau] \vdash s : \tau'$.

The subject reduction theorem tells us that β-reduction preserves the type of a term. This means that the reduction of a well-typed term cannot lead to a runtime type error.

Theorem 3.1.5 (Subject reduction)**.** $\Gamma \vdash t : \tau \wedge t \to_\beta t' \ \Rightarrow\ \Gamma \vdash t' : \tau$

This does not hold for β-expansion:

$$[x : \texttt{int}, y : \tau] \vdash y : \tau$$

and

$$y : \tau \quad \leftarrow_\beta \quad (\lambda z : \texttt{bool}.y)\ x$$

but: $(\lambda z : \texttt{bool}.y)\ x$ is not type-correct!

Theorem 3.1.6. *$\to_\beta$ ($\to_\eta$,$\to_{\beta\eta}$) over type-correct terms is confluent.*

This does not hold for all raw terms:

$$
\begin{tikzcd}
 & \lambda x : \texttt{int}.x \\
\lambda x : \texttt{int}.(\lambda y : \texttt{bool}.y)\ x \arrow[ur, "\beta"] \arrow[dr, "\eta"'] & \\
 & \lambda y : \texttt{bool}.y
\end{tikzcd}
$$

Theorem 3.1.7. *$\to_\beta$ terminates over type-correct terms.*

The proof is discussed in Section 3.2. A vague intuition is that the type system forbids self-application and thus recursion. This has the following positive consequence:

Corollary 3.1.8. *$=_\beta$ is decidable for type-correct terms.*

But there are type-correct terms s, such that the shortest reduction of s into a normal form has the length

$$\underbrace{2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}}_{\text{size of } s}.$$

However, these pathological examples are very rare in practice.

The negative consequence of Theorem 3.1.7 is the following:

Corollary 3.1.9. *Not all computable functions can be represented as type-correct $\lambda^{\to}$-terms.*

In fact, only polynomials + case distinction can be represented in $\lambda^{\to}$.

Question: Why are typed functional languages still Turing complete?

Theorem 3.1.10. *Assume we are given a base type nat with constants $0 : nat$, $succ : nat \to nat$, $pred : nat \to nat$, $ifz : nat \to nat \to nat \to nat$ and fixed-point combinators $Y_\tau : (\tau \to \tau) \to \tau$ for every type τ. Assume further that the constants come with the following reduction rules:*

$$\begin{array}{rcl@{\qquad\qquad}rcl} pred(succ\, t) & \to & t & ifz\, 0\, x\, y & \to & x \\ pred\, 0 & \to & 0 & ifz\,(succ\, t)\, x\, y & \to & y \\ Y_\tau\, t & \to & t\,(Y_\tau\, t) & & & \end{array}$$

Then every computable function can be represented as a closed type-correct $\lambda^{\to}$-term which contains as its only constants those introduced above.

3.2 Termination of $\to_\beta$

The proof in this section is based heavily on the combinatorial proof of Loader [Loa98]. A more general proof, which goes back to Tate, can also be found in Loader's notes or in the standard literature [HS08, GLT90, Han04].

For simplicity, we work with implicitly typed or even untyped terms.

Definition 3.2.1. Let t be an arbitrary λ-term. We say that t **diverges** (with regard to $\to_\beta$) if and only if there exists an infinite reduction sequence $t \to_\beta t_1 \to_\beta t_2 \to_\beta \cdots$. We say that t **terminates** (with regard to $\to_\beta$) and write $t\Downarrow$ if and only if t does not diverge.

We first define a subset T of untyped λ-terms:

$$\frac{r_1, \dots, r_n \in T}{x\, r_1 \dots r_n \in T}\ (Var) \qquad \frac{r \in T}{\lambda x. r \in T}\ (\lambda) \qquad \frac{r[s/x]\, s_1 \dots s_n \in T \quad s \in T}{(\lambda x. r)\, s\, s_1 \dots s_n \in T}\ (\beta)$$

Lemma 3.2.2. $t \in T \Rightarrow t\Downarrow$

Proof By induction on derivation of $t \in T$ ("rule induction").

(Var) $(x\, r_1 \dots r_n)\Downarrow$ follows directly from $r_1\Downarrow, \dots, r_n\Downarrow$, since x is a variable.

(λ) $(\lambda x. r)\Downarrow$ directly follows from $r\Downarrow$.

(β) Because of I.H. $(r[s/x]\, s_1 \dots s_n)\Downarrow$, $r\Downarrow$ and $s_i\Downarrow$, $i = 1, \dots, n$. If $(\lambda x. r)\, s\, s_1 \dots s_n$ diverged, there would have to exist the infinite reduction sequence of the following form:

$$(\lambda x. r)\, s\, s_1 \dots s_n \to_\beta^* (\lambda x. r')\, s'\, s_1' \dots s_n' \to_\beta r'[s'/x]\, s_1' \dots s_n' \to_\beta \cdots$$

since r, s (by I.H.) and all s_i terminate. However, $r[s/x]\, s_1 \dots s_n \to_\beta^* r'[s'/x]\, s_1' \dots s_n'$ also holds. This contradicts the termination of $r[s/x]\, s_1 \dots s_n$. Therefore $(\lambda x. r)\, s\, s_1 \dots s_n$ cannot diverge. $\square$

One can also show the converse. Thus T contains exactly the terminating terms.

Now we shall show that T is closed under substitution and application of type-correct terms. This is done by induction on the types. As we work with implicitly typed terms, the context Γ disappears. We simply write $t : \tau$.

We call a type τ **applicative** if and only if for all t, r and σ, the following holds.

$$\frac{t : \tau \to \sigma \quad r : \tau \quad t \in T \quad r \in T}{t\, r \in T}$$

We call τ **substitutive** if and only if for all s, r and σ, the following holds.

$$\frac{s : \sigma \quad r : \tau \quad x : \tau \quad s \in T \quad r \in T}{s[r/x] \in T}$$

Lemma 3.2.3. *Every substitutive type is applicative.*

Proof Let τ be substitutive. We show that τ is applicative by induction on the derivation of $t \in T$.

(Var) If $t = x\, r_1 \dots r_n$ and all $r_i \in T$, then $t\, r = x\, r_1 \dots r_n\, r \in T$ follows with (Var) since $r \in T$ by assumption.

(λ) If $t = \lambda x.s$ and $s \in T$, then $s[r/x] \in T$ holds since τ is substitutive. Therefore $t\, r = (\lambda x.s)r \in T$ follows with (β) since $r \in T$ by assumption.

(β) If $t = (\lambda x.u)\, s\, s_1 \dots s_n$ and $u[s/x]\, s_1 \dots s_n \in T$ and $s \in T$, then by I.H. $u[s/x]\, s_1 \dots s_n\, r \in T$ holds. Since $s \in T$, $t\, r = (\lambda x.u)\, s\, s_1 \dots s_n\, r \in T$ follows with (β). $\square$

Lemma 3.2.4. *Let $\tau = \tau_1 \to \dots \to \tau_k \to \tau'$, where τ' is not a function type. If all τ_i are applicative, then τ is substitutive.*

Proof by induction on the derivation of $s \in T$.

(Var) If $s = y\, s_1 \dots s_n$ and all $s_i \in T$, then $s_i[r/x] \in T$ holds by I.H., $i = 1, \dots, n$. If $x \neq y$, then $s[r/x] = y(s_1[r/x]) \dots (s_n[r/x]) \in T$ by (Var). If $x = y$, then $y : \tau$ holds, and therefore $s_i : \tau_i$, and $s_i[r/x] : \tau_i$, $i = 1, \dots, n$ as well. Since all τ_i are applicative, $s[r/x] = r(s_1[r/x]) \dots (s_n[r/x]) \in T$ holds.

(λ) If $s = \lambda y.u$ where $u \in T$, then by I.H. $u[r/x] \in T$. From this, $s[r/x] = \lambda y.(u[r/x]) \in T$ follows by (λ).

(β) If $s = (\lambda y.u)\, s_0\, s_1 \dots s_n$ by $u[s_0/y]\, s_1 \dots s_n \in T$ and $s_0 \in T$, then $s[r/x] = (\lambda y.(u[r/x]))$ $(s_0[r/x]) \dots (s_n[r/x]) \in T$ follows by (β) since $u[r/x][s_0[r/x]/y](s_1[r/x]) \dots (s_n[r/x])$ $= (u[s_0/y]\, s_1 \dots s_n)[r/x] \in T$ and $s_0[r/x] \in T$ by I.H. $\square$

Exercise 3.2.5. Show that for type-correct s and t the following holds: if $s \in T$ and $t \in T$ then $(s\ t) \in T$.

Theorem 3.2.6. *If t is type-correct, then $t \in T$ holds.*

Proof by induction on the derivation of the type of t. If t is a variable, then $t \in T$ holds by (Var). If $t = \lambda x.r$, then $t \in T$ follows by (λ) from the I.H. $r \in T$. If $t = r\, s$, then $t \in T$ follows by exercise 3.2.5 from the I.H. $r \in T$ and $s \in T$. $\square$

Theorem 3.1.7 is just a corollary of Theorem 3.2.6 and Lemma 3.2.2.

3.3 Type inference for $\lambda^{\to}$

Types:	τ	::=	`bool` | `int` | ...	basic types
			| α | β | γ | ...	type variables
			| $\tau_1 \to \tau_2$	

Terms: untyped λ-terms

Type inference rules:

$$\Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (t_1\ t_2) : \tau_2} \qquad \frac{\Gamma[x : \tau_1] \vdash t : \tau_2}{\Gamma \vdash (\lambda x.t) : \tau_1 \to \tau_2}$$

Terms can have distinct types (polymorphism):

$$\begin{array}{ll} \lambda x.x : & \alpha \to \alpha \\ \lambda x.x : & \mathtt{int} \to \mathtt{int} \end{array}$$

Definition 3.3.1. $\tau_1 \gtrsim \tau_2 \quad :\Leftrightarrow \quad \exists$ Substitution θ (of types for type variable) with $\tau_1 = \theta(\tau_2)$ ("τ_2 is more general than or equivalent to τ_1.")

Example:

$$\mathtt{int} \to \mathtt{int} \quad \gtrsim \quad \alpha \to \alpha \quad \gtrsim \quad \beta \to \beta \quad \gtrsim \quad \alpha \to \alpha$$

Every type-correct term has a most general type:

Theorem 3.3.2. $\Gamma \vdash t : \tau \quad \Rightarrow \quad \exists \sigma.\ \Gamma \vdash t : \sigma \ \wedge\ \forall \tau'. \Gamma \vdash t : \tau' \ \Rightarrow\ \tau' \gtrsim \sigma$

The proof idea (we do not go into the details) is to use the typing rules in a backward manner and generate constraints in the form of equations between types, much like a Prolog interpreter would apply the rules. We describe the generation of the constraints by adding an output parameter $\mid C$, a set of constraints, to our typing rules:

$$\frac{\Gamma(x) \text{ is defined}}{\Gamma \vdash x : \tau \mid \{\tau = \Gamma(x)\}}$$

$$\frac{\Gamma \vdash s : \tau_s \mid C_s \qquad \Gamma \vdash t : \tau_t \mid C_t}{\Gamma \vdash (s\ t) : \tau \mid \{\tau_s = \tau_t \to \tau\} \cup C_s \cup C_t} \qquad \frac{\Gamma[x : X] \vdash t : \tau' \mid C}{\Gamma \vdash \lambda x.t : \tau \mid \{\tau = X \to \tau'\} \cup C}$$

where X is a new type variable. The output C is computed by applying the rules backwards, starting with $\Gamma \vdash t : \tau$, where τ is typically a new type variable. In the end you obtain a set of constraints C such that $\Gamma \vdash t : \tau \mid C$. You now have to solve C (by unification) to obtain a substitution θ. If θ exists, $\theta(\tau)$ is a most general type of t (in context Γ).

Example, using Roman instead of Greek letters as type variables. We do not carry whole sets of constraints around but only note in each step which new costraint has been generated.

$$\begin{array}{ll} & \Gamma \vdash \lambda x.\lambda y.(y\ x) : A \\ \text{if} & [x : B] \vdash \lambda y.(y\ x) : C \text{ and } A = B \to C \\ \text{if} & [x : B, y : D] \vdash (y\ x) : E \text{ and } C = D \to E \\ \text{if} & [x : B, y : D] \vdash y : F \to E \quad \text{and} \quad [x : B, y : D] \vdash x : F \\ \text{if} & D = F \to E \quad \text{and} \quad B = F \end{array}$$

Therefore: $A = B \to C = F \to (D \to E) = F \to ((F \to E) \to E)$

Exercise 3.3.3. What is the set of constraints generated when trying to infer the type of $\lambda x.(x\ x)$? Does it have a solution?

3.4 let-polymorphism

Terms:

$$t \ ::= \ x \ \mid \ (t_1 \, t_2) \ \mid \ \lambda x.t \ \mid \ \texttt{let}\ x = t_1 \ \texttt{in}\ t_2$$

The intended meaning of $\texttt{let}\ x = t_1\ \texttt{in}\ \ t_2$ is $t_2[t_1/x]$. The meaning of a term with multiple lets is uniquely defined because of termination and confluence of $\to_{\underline{\beta}}$. We will now examine type inference in the presence of let.

Example:

$$\texttt{let} \quad \underbrace{f = \lambda x.x}_{f\,:\,\forall\alpha.\underbrace{\alpha\to\alpha}_{\tau}} \quad \texttt{in} \quad \underbrace{f}_{f\,:\,\tau[\alpha\to\alpha/\alpha]} \ \underbrace{f}_{f\,:\,\tau[\alpha/\alpha]}$$

Note

- $\forall$-quantified type variables can be replaced by arbitrary types.
- Although $(\lambda f.\mathrm{pair}\ (f\ 0)\,(f\ \mathrm{true}))\,(\lambda x.x)$ is semantically equivalent to the above let-term, it is not type-correct, because λ-bound variables do not have $\forall$-quantified types.

The grammar for types remains unchanged as in Section 3.3 but we add a new category of *type schemas* (σ):

$$\sigma \ ::= \ \forall\alpha.\sigma \ \mid \ \tau$$

Any type is a type schema. In general, type schemas are of the form $\forall\alpha_1 \dots \forall\alpha_n.\tau$, compactly written $\forall\alpha_1 \dots \alpha_n.\tau$.

Example of type schemas are α, $\texttt{int}$, $\forall\alpha.\alpha\to\alpha$ and $\forall\alpha\beta.\alpha\to\beta$. Note that $(\forall\alpha.\alpha\to\alpha)\to \texttt{bool}$ is not a type schema because the universal quantifier occurs inside a type.

The type inference rules now work with a context that associates type schemas with variable names: Γ is of the form $[x_1 : \sigma_1, \dots, x_n : \sigma_n]$:

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ (\mathrm{Var})$$

$$\frac{\Gamma \vdash t_1 : \tau_2 \to \tau \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1\ t_2) : \tau}\ (\mathrm{App})$$

$$\frac{\Gamma[x : \tau_1] \vdash t : \tau_2}{\Gamma \vdash (\lambda x.t) : \tau_1 \to \tau_2}\ (\mathrm{Abs})$$

$$\frac{\Gamma \vdash t_1 : \sigma_1 \qquad \Gamma[x : \sigma_1] \vdash t_2 : \sigma_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : \sigma_2}\ (\mathrm{Let})$$

Note that λ-bound variables have types (τ), let-bound variables have type schemas (σ).

Then there are the quantifier rules:

$$\frac{\Gamma \vdash t : \forall\alpha.\sigma}{\Gamma \vdash t : \sigma[\tau/\alpha]}\ (\forall\mathrm{Elim})$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash t : \forall\alpha.\sigma}\ (\forall\mathrm{Intro}) \qquad \text{if } \alpha \notin FV(\Gamma)$$

where $FV([x_1 : \sigma_1, \dots, x_n : \sigma_n]) = \bigcup_{i=1}^{n} FV(\sigma_i)$ and $FV(\forall\alpha_1 \dots \alpha_n.\tau) = Var(\tau) \setminus \{\alpha_1, \dots, \alpha_n\}$ and $Var(\tau)$ is the set of all type variables in τ.

Why does $(\forall\mathrm{Intro})$ need the condition $\alpha \notin FV(\Gamma)$?

Logic:	$x = 0 \vdash x = 0 \ \not\Rightarrow \ x = 0 \vdash \forall x.x = 0$
Programming:	$\lambda x.\texttt{let}\ y = x\ \texttt{in}\ y + (y\ 1)$ should not be type-correct.

But this term has a type if we drop the side-condition:

$$\dfrac{\dfrac{\dfrac{[x:\alpha]\vdash x:\alpha}{[x:\alpha]\vdash x:\forall\alpha.\alpha}\,(\forall\text{Intro}) \qquad \dfrac{\vdots}{[y:\forall\alpha.\alpha]\vdash y+(y\ 1):\texttt{int}}}{[x:\alpha]\vdash \texttt{let}\ y=x\ \texttt{in}\ y+(y\ 1):\texttt{int}}}{\lambda x.\texttt{let}\ y=x\ \texttt{in}\ y+(y\ 1):\alpha\to\texttt{int}}$$

Problem: The rules do not provide any algorithm, since quantifier rules are not syntax-directed, i.e. they are (almost) always applicable.

Solution: Integrate ($\forall$Elim) with (Var) and ($\forall$Intro) with (Let):

$$\dfrac{\Gamma(x)=\forall\alpha_1\ldots\alpha_n.\tau}{\Gamma\vdash x:\tau[\tau_1/\alpha_1,\ldots,\tau_n/\alpha_n]}\ (\text{Var'})$$

$$\dfrac{\Gamma\vdash t_1:\tau \qquad \Gamma[x:gen(\Gamma,\tau)]\vdash t_2:\tau_2}{\Gamma\vdash \texttt{let}\ x=t_1\ \texttt{in}\ t_2:\tau_2}\ (\text{Let'})$$

where $gen(\Gamma,\tau)=\forall\alpha_1,\ldots,\alpha_n.\tau$ and $\{\alpha_1,\ldots,\alpha_n\}=Var(\tau)\setminus FV(\Gamma)$.

Rules (Var) and (Let) are replaced by (Var') and (Let'); (App) and (Abs) remain unchanged; ($\forall$Intro) and ($\forall$Elim) disappear. The resulting system has four syntax-directed rules and all typing judgements are of the form $\Gamma\vdash t:\tau$; type schemas occur only in Γ.

Example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{D=F*E}{\Gamma'\vdash p:F\to(E\to D)} \quad \dfrac{F=A}{\Gamma'\vdash x:F}}{\Gamma'\vdash p\ x:E\to D} \quad \dfrac{C=E}{\Gamma'\vdash z:E}}{\Gamma'\vdash(p\ x)\ z:D}}{\Gamma[x:A]\vdash\lambda z.p\ x\ z:C\to D} \quad \dfrac{\dfrac{B=A*G}{\Gamma''\vdash y:G\to B} \quad \dfrac{\dfrac{G=A*\texttt{int}}{\Gamma''\vdash y:H\to G} \quad \dfrac{H=\texttt{int}}{\Gamma''\vdash 1:H}}{\Gamma''\vdash y\ 1:G}}{\Gamma''\vdash y\ (y\ 1):B}}{\Gamma[x:A]\vdash\texttt{let}\ y=\lambda z.p\ x\ z\ \texttt{in}\ y\ (y\ 1):B}}{\Gamma=[1:\texttt{int},p:\forall\alpha,\beta.\alpha\to\beta\to(\alpha*\beta)]\vdash\lambda x.\texttt{let}\ y=\lambda z.p\ x\ z\ \texttt{in}\ y\ (y\ 1):A\to B}$$

(where $\Gamma'=\Gamma[x:A,z:C]$ and $\Gamma''=\Gamma[x:A,y:\forall C.C\to A*C]$)

$\Rightarrow\ B=A*(A*\texttt{int})$

Let DM be the system with the rules (Var), (App), (Abs), (Let), ($\forall$Elim) and ($\forall$Intro) and DM' the system with the rules (Var'), (App), (Abs) and (Let'). Because each rule in DM' can be simulated in DM we have:

Lemma 3.4.1. *$\Gamma\vdash_{DM'} t:\tau\ \Rightarrow\ \Gamma\vdash_{DM} t:\tau$.*

To state the opposite direction we need a definition of "more general" on type schemas:

$$\forall\overline{\alpha_m}.\tau\preceq\forall\overline{\beta_n}.\tau' \ \text{ iff } \ \exists\overline{\tau_m}.\ \tau'=\tau[\overline{\tau_m}/\overline{\alpha_m}]\wedge\beta_1,\ldots,\beta_n\notin FV(\forall\overline{\alpha_m}.\tau)$$

Theorem 3.4.2. *$\Gamma\vdash_{DM} t:\sigma\ \Rightarrow\ \exists\tau.\ \Gamma\vdash_{DM'} t:\tau\wedge gen(\Gamma,\tau)\preceq\sigma$*

Complexity of type inference:

- without `let`: linear

- with `let`: DEXPTIME-complete (Types can grow exponentially with the size of the terms.)

Example:

$$
\begin{array}{l}
\texttt{let}\ x_0 = \lambda y.\lambda z.z\ y\ y \\
\texttt{in}\ \texttt{let}\ x_1 = \lambda y.x_0\ (x_0\ y) \\
\quad \texttt{in}\ \ldots \\
\qquad \ddots \\
\qquad\quad \texttt{let}\ x_{n+1} = \lambda y.x_n\ (x_n\ y) \\
\qquad\quad \texttt{in}\ x_{n+1}\ (\lambda z.z)
\end{array}
$$

Chapter 4

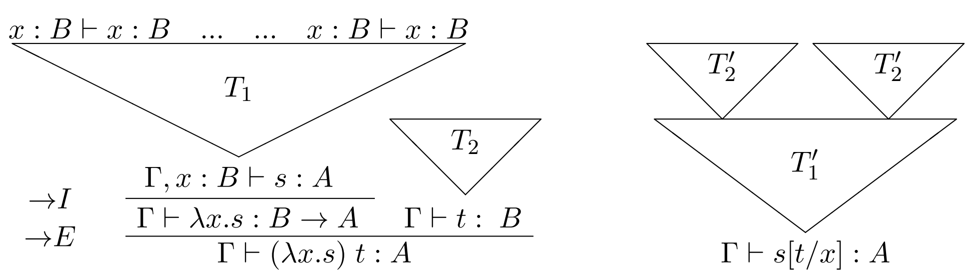
The Curry-Howard Isomorphism

4.1 Simply Typed λ-Calculus

typed λ-calculus ($\lambda^{\rightarrow}$)	constructive logic (intuitionistic propositional logic)
Types: $\tau \ ::= \ \alpha \,\vert\, \beta \,\vert\, \gamma \,\vert\, \ldots \,\vert\, \tau \to \tau$	Formulas: $A \ ::= \ \underbrace{P \,\vert\, Q \,\vert\, R \,\vert\, \ldots}_{\text{propositional variable}} \ \vert\, A \to A$
$\Gamma \vdash t : \tau$	$\Gamma \vdash A$ (Γ: finite set of formulas)
$\dfrac{\Gamma \vdash t_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1\ t_2) : \tau_1}$ (App) $\dfrac{\Gamma[x : \tau_1] \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}$ (Abs) $\Gamma \vdash x : \Gamma(x)$ if $\Gamma(x)$ is defined	$\dfrac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$ ($\to$ Elim) $\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$ ($\to$ Intro) $\Gamma \vdash A$ if $A \in \Gamma$
type-correct λ-terms	proofs
Example: $\dfrac{[x : \alpha] \vdash x : \alpha}{\vdash \lambda x.x : \alpha \to \alpha}$ The λ-term encodes the skelton of the proof.	$\dfrac{A \vdash A}{\vdash A \to A}$ This derivation is represented in a compact manner by $\lambda x.x$ and can be reconstructed by type inference.

For brevity we write $\to I/E$ instead of $\to$ Intro/Elim in the sequel.

Proofs where the first premise of $\to E$ proved by $\to I$ can be reduced:



Proof reduction = Lemma-elimination
Correctness follows from subject reduction: types are invariant under β-reduction.

Example:

$$\underbrace{(\underbrace{(A \to A)}_{a'} \to B \to C)}_{x} \to \underbrace{((A \to A) \to B)}_{y} \to C \;=:\; \phi$$

Two proofs:

$$\begin{array}{lll} & \lambda x.\lambda y.(\lambda a'.x\ a'\ (y\ a'))\ (\lambda a.a) : \phi & \text{proof by lemma } A \to A \\ \longrightarrow & \lambda x.\lambda y.x\ (\lambda a.a)\ (y\ (\lambda a.a)) : \phi & \text{proof in normal form} \end{array}$$

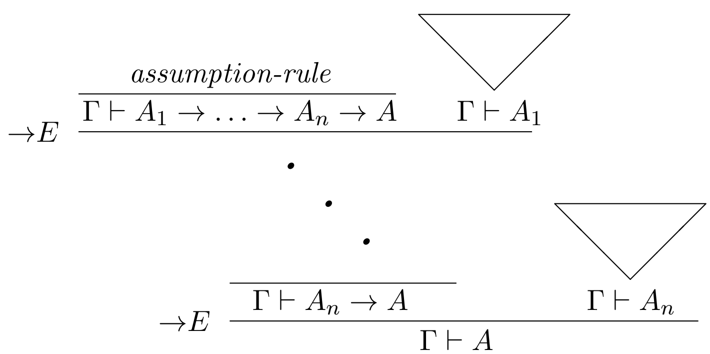
Definition 4.1.1. A proof is in **normal form** if the corresponding λ-term is in normal form.

Therefore a proof is in normal form if and only if no part of the proof has the following form, introduction followed by elimination:

$$\to E\,\dfrac{\to I\,\dfrac{\cdots}{\cdots} \qquad \cdots}{\cdots}$$

A typed λ-term in normal form that is not a λ-abstraction must be of the form $x\ t_1 \cdots t_n$. Translating this into the language of proofs it means:

Lemma 4.1.2. *A proof in normal form that does not end with $\to I$ has to have the following form:*



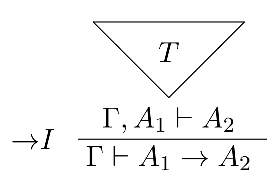
In the sequel note that every formula is a subformula of itself.

Theorem 4.1.3. *In a proof of $\Gamma \vdash A$ in normal form, only subformulas of Γ and A occur. This is called the **subformula property**.*

Proof: by induction on the derivation of $\Gamma \vdash A$.

1. $\Gamma \vdash A$ with $A \in \Gamma$: obvious

2.



Induction hypothesis: only subformulas of Γ, A_1 and A_2 occur in T. Hence the assertion follows immediately.

3. If the last rule is $\to E$, Lemma 4.1.2 applies. Because of assumption-rule: $A_1 \to \cdots \to A_n \to A \in \Gamma$. Ind. hyp. for the subproofs $\Gamma \vdash A_i$: only subformulas of Γ and A_i occur and thus only subformulas of Γ. In the leftmost branch of the proof only Γ and subformulas of $A_1 \to \cdots \to A_n \to A$ occur. Therefore, in the whole tree only subformulas of Γ occur. □

Theorem 4.1.4. $\Gamma \vdash A$ *is decidable.*

The proof is the following algorithm:

Search for proof tree in normal form (always exists, since $\to_\beta$ terminates for type-correct terms) by building it up from the root to the leaves. The algorithm is expressed as the following recursive function $prove(\Gamma \vdash A)$ that may succeed (with a proof of $\Gamma \vdash A$) or fail:
Cycle test: if this call of $prove(\Gamma \vdash A)$ is a descendant of a previous call of $prove(\Gamma \vdash A)$, then fail. Otherwise try to prove $\Gamma \vdash A$:
If $\Gamma \vdash A$ with $A \in \Gamma$, then succeed with proof by assumption.
Otherwise try to use $\to I$, if $A = B \to C$, and call $prove(\Gamma, B \vdash C)$. If this fails or A is not an implication, try to use $\to E$ as in Lemma 4.1.2: try all $A_1 \to \cdots \to A_n \to A \in \Gamma$ (one after the other, finite choice) and for each one call $prove(\Gamma \vdash A_i)$ for all $i = 1, \ldots, n$.
This algorithm terminates for the following reasons. A call $prove(\Gamma \vdash A)$ can generate only direct recursive calls $prove(\Gamma' \vdash A')$ where all the formulas in Γ', A' are subformulas of formulas in Γ, A. By transitivity of the subformula relation, this is also true for indirect recursive calls. Thus there are only finitely many possible arguments for all recursive calls of $prove(\Gamma \vdash A)$. Because of the cycle test, all calls must terminate. □

Example:

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash P \to Q \to R \quad \Gamma \vdash P}{\Gamma \vdash Q \to R}\to E \qquad \cfrac{\Gamma \vdash P \to Q \quad \Gamma \vdash P}{\Gamma \vdash Q}\to E}{\Gamma := P \to Q \to R, P \to Q, P \vdash R}\to E}{\vdash (P \to Q \to R) \to (P \to Q) \to P \to R}\text{3 times} \to I$$

Peirce's law $((P \to Q) \to P) \to P$ is not provable in intuitionistic logic. Note that $\vdash \phi$ is never provable by $\to E$ because that would require a formula $A_1 \to \cdots \to A_n \to \phi$ in the context but the context is empty. Hence we try proof by $\to I$:

$$\cfrac{\cfrac{\Gamma \vdash A_n \to P \quad \Gamma \vdash A_n}{\Gamma := (P \to Q) \to P \vdash P}\to E}{\vdash ((P \to Q) \to P) \to P}\to I$$

with $A_1 \to \cdots \to A_n \to P \in \Gamma \;\Rightarrow\; n = 1$ and $A_n = P \to Q$. Consider $\Gamma \vdash P \to Q$. The derivation cannot be done by $\to E$, because Γ does not contain any formula of the form $\cdots \to (P \to Q)$. Hence:

$$\cfrac{\cfrac{\Gamma, P \vdash B_n \to Q \quad \Gamma, P \vdash B_n}{\Gamma, P \vdash Q}\to E}{\Gamma \vdash P \to Q}\to I$$

with $B_1 \to \cdots \to B_n \to Q \in \Gamma, P$ — but such a formula is not found in Γ and P. Thus Peirce's law is not provable.

Note that Peirce's law is a tautologie in classical two-valued propositional logic. Therefore constructive logic is incomplete with regard to two-valued models. There are alternative, more complicated notions of models for intuitinistic logic. The decision problem if a propositional formula is a tautology is NP-complete for classical two-valued logic but PSPACE-complete for intuitionistic logic.

Exercise 4.1.5. Prove $\vdash ((((p \to q) \to p) \to p) \to q) \to q$.

Exercise 4.1.6. The algorithm in Theorem 4.1.4 can be streamlined as follows:

1. When trying to prove $\Gamma \vdash A \to B$, it suffices to try $\to I$. Explain why.
2. The attempt to prove $\Gamma \vdash A$ by assumption can be dropped: it is subsumed by the alternative using Lemma 4.1.2. However, the proof obtained can be different. Explain the difference and why the outright proof by assumption is subsumed.

Here are two examples that go beyond propositional logic but illustrate the fundamental difference between constructive and not-constructive proofs:

1. $\forall k \geq 8. \exists m, n.\, k = 3m + 5n$

   Proof: by induction on k.

   Base case: $k = 8 \Rightarrow (m, n) = (1, 1)$

   Step: Assume $k = 3m + 5n$ (induction hypothesis)

   Case distinction:

   1. $n \neq 0 \;\Rightarrow\; k + 1 = (m + 2) * 3 + (n - 1) * 5$
   2. $n = 0 \;\Rightarrow\; m \geq 3 \;\Rightarrow\; k + 1 = (m - 3) * 3 + (n + 2) * 5$ □

   Corresponding algorithm:

   $$\begin{array}{l} f : \mathbb{N}_{\geq 8} \to \mathbb{N} \times \mathbb{N} \\ f(8) = (1,1) \\ f(k+1) = \texttt{let}\ (m,n) = f(k) \\ \qquad\qquad\quad \texttt{in if}\ n \neq 0\ \texttt{then}\ (m+2, n-1)\ \texttt{else}\ (m-3, n+2) \end{array}$$

2. $\exists$ irrational $a, b.\ a^b$ is rational.

   Case distinction:

   1. $\sqrt{2}^{\sqrt{2}}$ rational $\Rightarrow a = b = \sqrt{2}$
   2. $\sqrt{2}^{\sqrt{2}}$ irrational $\Rightarrow a = \sqrt{2}^{\sqrt{2}}, b = \sqrt{2} \;\Rightarrow\; a^b = \sqrt{2}^{2} = 2$

Classification:

Question	Types	Formulas
$t : \tau$? (t explicitly typed)	Does t have the type τ ?	Is t a correct proof of formula τ ?
$\exists \tau. t : \tau$	type inference	What does the proof t prove?
$\exists t. t : \tau$	program synthesis	proof search

4.2 More Propositional Logic

4.2.1 Conjunction = Cartesian Product

We extend terms with pairs and projections:

$$t \quad ::= \quad \dots \quad | \quad <t_1, t_2> \quad | \quad \pi_1\, t \quad | \quad \pi_2\, t$$

New proof rules for new connective "$\wedge$":

$$\frac{\Gamma \vdash t_1 : A_1 \quad \Gamma \vdash t_2 : A_2}{\Gamma \vdash <t_1, t_2> : A_1 \wedge A_2}\ \wedge I \qquad \frac{\Gamma \vdash p : A_1 \wedge A_2}{\Gamma \vdash \pi_i\, p : A_i}\ \wedge E_i\ (i = 1, 2)$$

Conjunction behaves like Cartesian product: $\wedge \approx \times$.

Example proof:

$$\dfrac{\dfrac{\dfrac{p : A \wedge B \vdash p : A \wedge B}{p : A \wedge B \vdash \pi_2\, p : B}\ \wedge E_2 \quad \dfrac{p : A \wedge B \vdash p : A \wedge B}{p : A \wedge B \vdash \pi_1\, p : A}\ \wedge E_1}{p : A \wedge B \vdash <\pi_2\, p, \pi_1\, p> : B \wedge A}\ \wedge I}{\vdash \lambda p. <\pi_2\, p, \pi_1\, p> : A \wedge B \to B \wedge A}\ \to I$$

Reduction rules:

$$\pi_i <t_1, t_2> \to_{\pi_i} t_i \quad (i = 1, 2)$$

As proof reductions:

$$\dfrac{\dfrac{\dfrac{D_1}{\Gamma \vdash t_1 : A_1} \quad \dfrac{D_2}{\Gamma \vdash t_2 : A_2}}{\Gamma \vdash <t_1, t_2> : A_1 \wedge A_2}\ \wedge I}{\Gamma \vdash \pi_i <t_1, t_2> : A_i}\ \wedge E_i \qquad \to_{\pi_i} \qquad \dfrac{D_i}{\Gamma \vdash t_i : A_i}$$

Eliminates $\wedge I$ followed by $\wedge E$.

Theorem 4.2.1. *The joint reduction relation $\to_\beta \cup \to_{\pi_i}$ is terminating on type-correct terms.*

Theorem 4.2.2. *The joint reduction relation $\to_\beta \cup \to_{\pi_i}$ is confluent on type-correct terms.*

Proof idea: there is no "critical" overlap between rules, thus the reduction is locally confluent and by Newman's Lemma confluent.

Theorem 4.2.3. *Proofs in normal form have the subformula property and $\Gamma \vdash A$ is decidable.*

Proof: a simple extensions of the corresponding proofs for $\to$ alone.

We now look briefly at a further extension, **surjective pairing**. We only consider terms and reductions.

$$<\pi_1\, t, \pi_2\, t> \quad \to_{sp} \quad t$$

By $\to$ we abbreviate $\to_\beta \cup \to_{\pi_i} \cup \to_{sp}$. Similar to above, one can prove that $\to$ is terminating and confluent on type-correct terms. The proof of local confluence is a little bit more interesting because this time there is a critical overlap, but that overlap is trivial because both reducts are identical:

$$\begin{array}{ccc} & & \pi_i\, t \\ & \nearrow_{\pi_i} & \\ \pi_i <\pi_1\, t, \pi_2\, t> & & \\ & \searrow_{sp} & \\ & & \pi_i\, t \end{array}$$

Thus $\to$ is locally confluent and hence confluent on type-correct terms.

However, it should be noted that $\to$ is not confluent on all terms (incl. not type-correct terms like Y): Let $r = (Y\, s)$, $s = (Y\, t)$ and $t = \lambda xy.\, <\pi_1(zy), \pi_2(z(xy))>$. Then $r \to^*_{\beta sp} z(sr)$ and $r \to^*_{\beta sp} t(z(sr)$ but these two terms have no common reduct.

We do not consider surjective pairing in the sequel.

4.2.2 Disjunction = Disjoint Union

We extend terms with case-expressions and injections:

$$t \quad ::= \quad \dots \quad | \quad in_1\, t \quad | \quad in_2\, t \quad | \quad case\, t\, t_1\, t_2$$

Notation:

- Frequent alternative names for in_1/in_2 are inl/inr.
- Syntactic sugar: $case\ t\ of\ in_1\, x \Rightarrow t_1 \mid in_2\, y \Rightarrow t_2 \ \equiv\ case\ t\ (\lambda x.\, t_1)\ (\lambda y.\, t_2)$

New proof rules for new connective "$\vee$":

$$\frac{\Gamma \vdash t : A_i}{\Gamma \vdash in_i\, t : A_1 \vee A_2}\ \vee I_1\ (i = 1, 2) \qquad \frac{\Gamma \vdash t : A_1 \vee A_2 \quad \Gamma, x{:}A_1 \vdash t_1 : B \quad \Gamma, y{:}A_2 \vdash t_2 : B}{\Gamma \vdash case\, t\, (\lambda x.\, t_1)\, (\lambda y.\, t_2) : B}\ \vee E$$

Disjunction behaves like disjoint union: $\vee \approx +$.

Reduction rules:

$$case\ (in_i\, t)\ t_1\ t_2 \quad \to_{in_i} \quad (t_i\, t) \qquad (i = 1, 2)$$

On the level of types/formulas:

$$\frac{\dfrac{\dfrac{D}{\Gamma \vdash A_i}}{\Gamma \vdash A_1 \vee A_2}\ \vee I_i \quad \dfrac{D_1}{\Gamma, A_1 \vdash B} \quad \dfrac{D_2}{\Gamma, A_2 \vdash B}}{\Gamma \vdash B}\ \vee E \quad \to_{in_i} \quad \frac{\dfrac{\dfrac{D_i}{\Gamma, A_i \vdash B}}{\Gamma \vdash A_i \to B}\to I \quad \dfrac{D}{\Gamma \vdash A_i}}{\Gamma \vdash B}\to E$$

Eliminates $\vee I$ followed by $\vee E$.

Theorem 4.2.4. *The reduction relation $\to_\beta \cup \to_{\pi_i} \cup \to_{in_i}$ is terminating and confluent.*

But proofs in normal form do not have the subformula propertty:

$$\frac{\dfrac{\dfrac{}{A \vee A \vdash A \vee A} \quad \dfrac{\dfrac{}{A \vdash A} \quad \dfrac{}{A \vdash A}}{A \vdash A \wedge A}\wedge I \quad \dfrac{\dfrac{}{A \vdash A} \quad \dfrac{}{A \vdash A}}{A \vdash A \wedge A}\wedge I}{A \vee A \vdash A \wedge A}\vee E}{A \vee A \vdash A}\wedge E$$

New reduction rules:

$$\pi_i\ case\ t\ (\lambda x.\, t_1)\ (\lambda y.\, t_2) \quad \to \quad case\ t\ (\lambda x.\, \pi_i\, t_1)\ (\lambda y.\, \pi_i\, t_2)$$

Same problem with $\dfrac{\vee E}{\to E}$ and $\dfrac{\vee E}{\vee E}$. A possible reduction for $\dfrac{\vee E}{\to E}$:

$$(case\ t\ (\lambda x.\, t_1)\ (\lambda y.\, t_2))\, u \quad \to \quad case\ t\ (\lambda x.\, t_1\, u)\ (\lambda y.\, t_2\, u)$$

All required reductions (known as **commuting conversions**) can be expressed uniformly by one rule schema:

$$E[\mathit{case}\ t\ (\lambda x.\, t_1)\ (\lambda y.\, t_2)] \rightarrow \mathit{case}\ t\ (\lambda x.\, E[t_1])\ (\lambda y.\, E[t_2]) \tag{4.1}$$

where $E[.]$ is an "elimination context", i.e. the **principal premise** of an elimination rule, i.e. the premise with the connective that is eliminated. Thus there are three possible cases for $E[e]$: $\pi_i\, e$, $(s\ e)$ and $\mathit{case}\ e\ (\lambda v.\ s_1)\ (\lambda w.\ s_2)$. The resulting reduction relation is still terminating and confluent and posesses the subformula property.

Exercise 4.2.5. Write the three commuting conversions (4.1) as reduction rules between proof trees, without λ-terms, for example like the second version of $\rightarrow_{in_i}$ above.

4.2.3 False and Negation

Syntax for False: $\bot$.

Proof rule:

$$\frac{\Gamma \vdash t : \bot}{\Gamma \vdash \varepsilon\, t : A}\ \bot E$$

Think ε = "exception" or "error". There is no introduction rule!

We consider $\neg A$ as an abbreviation of $A \rightarrow \bot$. This leads to the following derived rules:

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg I \qquad \frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \bot}\ \neg E$$

Classical two-valued logic adds

$$\frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ \mathit{contr}$$

For confluence, more reduction rules are needed: $E[\varepsilon\, t] \rightarrow \varepsilon\, t$

4.3 System F

Terms and types:

$$\begin{array}{rcl} t & ::= & x \mid c \mid (t_1\ t_2) \mid \lambda x : \tau.\, t \mid \lambda \alpha.\, t \mid (t\ \tau) \\ \tau & ::= & \alpha \mid \tau_1 \rightarrow \tau_2 \mid \forall \alpha.\, \tau \end{array}$$

Examples: $\lambda \alpha.\, \lambda x : \alpha.\, x$ which has type $\forall \alpha.\, \alpha \rightarrow \alpha$
$(((\lambda \alpha.\, \lambda x : \alpha.\, x)\ \mathit{int})\ 5)$

Remarks:

- Important application: programming languages with powerful type systems and explicit type annotations. System F is the basis of Haskell's intermediate language Core. Types are not needed at runtime but are required for type-checking.
- Generalizes ML types. Example: $(\forall \alpha.\, \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$

Proof rules: $\lambda^{\rightarrow}$ extended with

$$\frac{\Gamma \vdash t : \tau \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash \lambda \alpha.\, t : \forall \alpha.\, \tau}\ \forall I \qquad \frac{\Gamma \vdash t : \forall \alpha.\, \tau}{\Gamma \vdash (t\ \tau') : \tau[\tau'/\alpha]}\ \forall E$$

Example: let $\tau = \forall\alpha.\, \alpha \to \alpha$:

$$\dfrac{\dfrac{x:\tau \vdash x:\tau}{x:\tau \vdash (x\ \tau):\tau\to\tau}\ \forall E \qquad x:\tau\vdash x:\tau}{\dfrac{x:\tau\vdash ((x\ \tau)\ x):\tau}{\vdash \lambda x:\tau.\,((x\ \tau)\ x):\tau\to\tau}\ {\to}I}\ {\to}E$$

Reduction rules: $\to_\beta$ and

$$((\lambda\alpha.\, t)\ \tau) \to t[\tau/\alpha]$$

Theorem 4.3.1. *On type-correct terms, the reduction relation is confluent and terminating.*

Proofs in normal form obey a subformula property where $\tau[\tau'/\alpha]$ is considered a subformula of $\forall\alpha.\,\tau$. Thus it does not follow that we can decide if a type is inhabited, i.e. if there is a term of that type. In fact:

Theorem 4.3.2. *In System F it is not decidable if a given type is inhabited.*

Nor is type inference decidable:

Theorem 4.3.3 (Wells)**.** *It is undecidable if for a given untyped λ-term t there is a type-correct term t' in System F such that erasing all types from t' yields t.*

We will now see how we can *define* data types and propositional logic in System F.

4.3.1 Booleans

Typed version of untyped construction.

$bool = \forall\alpha.\alpha\to\alpha\to\alpha$
$true = \lambda\alpha.\lambda x:\alpha.\lambda y:\alpha.\, x \quad (: bool)$
$false = \lambda\alpha.\lambda x:\alpha.\lambda y:\alpha.\, y \quad (: bool)$
$if = \lambda b: bool.\, b$

Typing of *if*: $b:bool, x:\tau, y:\tau \vdash if\ b\ \tau\ x\ y:\tau$
Reductions as required: $if\ true\ \tau\ s\ t \to true\ \tau\ s\ t \to^3 s$

4.3.2 Conjunction

Typed version of untyped construction.

$\tau_1\times\tau_2 = \forall\alpha.(\tau_1\to\tau_2\to\alpha)\to\alpha$ (container waiting for consumer)
$pair = \lambda\alpha.\lambda\beta.\lambda x:\alpha.\lambda y:\beta.\lambda\gamma.\lambda p:\alpha\to\beta\to\gamma.\,(p\ x)\ y \quad (:\forall\alpha\forall\beta.\,\alpha\to\beta\to\alpha\times\beta)$
$\pi_i = \lambda\alpha_1.\lambda\alpha_2.\lambda p:\alpha_1\times\alpha_2.\, p\ \alpha_i\ (\lambda x_1:\alpha_1.\lambda x_2:\alpha_2.\, x_i) \quad (:\forall\alpha_1.\forall\alpha_2.\,\alpha_1\times\alpha_2\to\alpha_i)$

Reductions as required: $\pi_i\ \tau_1\ \tau_2\ (pair\ \tau_1\ \tau_2\ t_1\ t_2) \to^* t_i$

Interpreting $\times$ as $\wedge$, the definition of $\wedge$ is

$$A_1\wedge A_2 = \forall B.\,(A_1\to A_2\to B)\to B$$

From this definition we can prove $\wedge E_i$:

$$\dfrac{\dfrac{\Gamma\vdash A_1\wedge A_2}{\Gamma\vdash (A_1\to A_2\to A_i)\to A_i}\ \forall E \qquad \dfrac{\vdots}{\Gamma\vdash A_1\to A_2\to A_i}}{\Gamma\vdash A_i}\ {\to}E$$

Lemma 4.3.4 (Weakening). *If* $\Gamma \subseteq \Gamma'$ *and* $\Gamma \vdash A$ *then* $\Gamma' \vdash A$

The proof is by induction on $\Gamma \vdash A$.

Exercise 4.3.5. Derive $\wedge I$ considering only formulas, not terms, like in the derivation of $\wedge E_i$.

4.3.3 Recursive Types

Motivation:

$$\texttt{data}\ T\ \overline{\alpha} = C_1\ \tau_{11} \dots \tau_{1n_1} \mid \cdots \mid C_k\ \tau_{k1} \dots \tau_{kn_k}$$

Restriction: if T occurs in τ_{ij} then $\tau_{ij} = T\ \overline{\alpha}$.

Satisfies restriction:

```
data Prod a b = Pair a b
data Sum a b = In1 a | In2 b
data Nat = Z | S Nat
data List a = Nil | Cons a (List a)
```

Not covered but legal Haskell:

```
data T a = C (T (a,a))
data T = C (T -> T)
```

Representation of the data type

$\tau_i = \tau_{i1} \to \cdots \to \tau_{in_i} \to T\ \overline{\alpha}$ (Type of C_i)
$\sigma_i = \tau_i[\gamma / T\ \overline{\alpha}]$
$T\ \overline{\alpha} = \forall \gamma.\, \sigma_1 \to \cdots \sigma_k \to \gamma$
where γ is a new type variable.

Example: $\mathit{Sum}\ \alpha\ \beta = \forall \gamma.\, (\alpha \to \gamma) \to (\beta \to \gamma) \to \gamma$

Definition of the constructors

$C_i = \lambda \overline{\alpha}. \lambda x_1 : \tau_{i1} \dots \lambda x_{n_i} : \tau_{in_i}. \lambda \gamma. \lambda f_1 : \sigma_1 \dots \lambda f_k : \sigma_k.\, f_i\ s_1 \dots s_{n_i}$

where $s_j = \begin{cases} x_j & \text{if } \tau_{ij} \neq T\ \overline{\alpha} \\ x_j\ \gamma\ f_1 \dots f_k & \text{otherwise} \end{cases}$

Example: $\mathit{In1} = \lambda \alpha \beta.\, \lambda x : \alpha.\, \lambda \gamma.\, \lambda f : \alpha \to \gamma.\, \lambda g : \beta \to \gamma.\, f\ x$

Definition of primitive recursor

Specification:

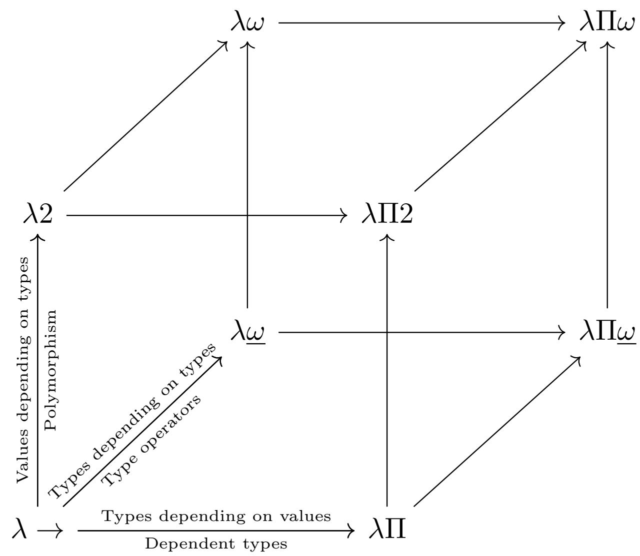
$\mathit{rec} : \forall \overline{\alpha}.\, T\ \overline{\alpha} \to T\ \overline{\alpha}$
$\mathit{rec}\ \overline{\rho}\ (C_i\ x_1 \dots x_{n_i})\ \sigma\ f_1 \dots f_k = f_i\ t_1 \dots t_{n_i}$
where $t_j = \begin{cases} x_j & \text{if } \tau_{ij} \neq T\ \overline{\alpha} \\ \mathit{rec}\ \overline{\rho}\ x_j\ \sigma\ f_1 \dots f_k & \text{otherwise} \end{cases}$

Implementation:

$\mathit{rec} = \lambda \overline{\alpha}.\, \lambda x : T\ \overline{\alpha}.\, x$

Unifies *if*, *case* and π_i

4.4 Barendregt's Lambda Cube



Note: $\lambda 2$ = System F and $\lambda\Pi\omega$ = Calculus of Constructions

Appendix A

Relational Basics

A.1 Notation

In the following, $\rightarrow \subseteq A \times A$ is an arbitrary binary relation over a set A. Instead of $(a, b) \in \rightarrow$ we write $a \rightarrow b$.

Definition A.1.1.

$$
\begin{array}{lcll}
x \stackrel{=}{\rightarrow} y & :\Leftrightarrow & x \rightarrow y \vee x = y & \text{(reflexive closure)} \\
x \leftrightarrow y & :\Leftrightarrow & x \rightarrow y \vee y \rightarrow x & \text{(symmetric closure)} \\
x \stackrel{n}{\rightarrow} y & :\Leftrightarrow & \exists x_1, \ldots, x_n.\ x = x_1 \rightarrow x_2 \rightarrow \cdots \rightarrow x_n = y & \\
x \stackrel{+}{\rightarrow} y & :\Leftrightarrow & \exists n > 0.\ x \stackrel{n}{\rightarrow} y & \text{(transitive closure)} \\
x \stackrel{*}{\rightarrow} y & :\Leftrightarrow & \exists n \geq 0.\ x \stackrel{n}{\rightarrow} y & \text{(reflexive and transitive closure)} \\
x \stackrel{*}{\leftrightarrow} y & :\Leftrightarrow & x\,(\leftrightarrow)^*\,y & \text{(reflexive, transitive and symmetric closure)}
\end{array}
$$

Definition A.1.2. An element a is in **normal form wrt.** $\rightarrow$ if these does not exists any b that satisfies $a \rightarrow b$.

A.2 Confluence

Definition A.2.1. A relation $\rightarrow$

is **confluent**, if $x \stackrel{*}{\rightarrow} y_1 \ \wedge x \stackrel{*}{\rightarrow} y_2 \ \Rightarrow \ \exists z.\ y_1 \stackrel{*}{\rightarrow} z \ \wedge \ y_2 \stackrel{*}{\rightarrow} z$.

is **locally confluent**, if $x \rightarrow y_1 \ \wedge x \rightarrow y_2 \ \Rightarrow \ \exists z.\ y_1 \stackrel{*}{\rightarrow} z \ \wedge \ y_2 \stackrel{*}{\rightarrow} z$.

has the **diamond-property**, if $x \rightarrow y_1 \ \wedge x \rightarrow y_2 \ \Rightarrow \ \exists z.\ y_1 \rightarrow z \ \wedge \ y_2 \rightarrow z$.

$$
\begin{tikzcd}
x \arrow[r, "*"] \arrow[d, "*"'] & y_1 \arrow[d, dashed, "*"'] \\
y_2 \arrow[r, dashed, "*"] & z
\end{tikzcd}
\qquad
\begin{tikzcd}
x \arrow[r] \arrow[d] & y_1 \arrow[d, dashed, "*"'] \\
y_2 \arrow[r, dashed, "*"] & z
\end{tikzcd}
\qquad
\begin{tikzcd}
x \arrow[r] \arrow[d] & y_1 \arrow[d, dashed] \\
y_2 \arrow[r, dashed] & z
\end{tikzcd}
$$

confluence local confluence the diamond-property

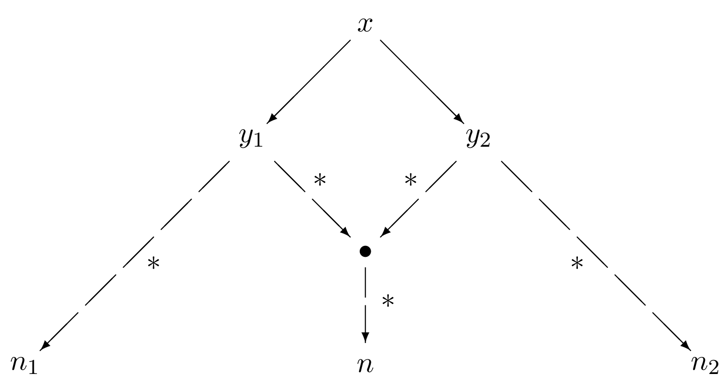
Figure A.1: Sketch of Definition A.2.1

Fact A.2.2. *If $\to$ is confluent, then every element has at most one normal form.*

Lemma A.2.3 (Newman's Lemma)**.** *If $\to$ is locally confluent and terminating, then $\to$ is confluent.*

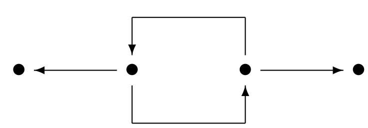
Proof: by contradiction

Assumption: $\to$ is not confluent, i.e. there is an x with two distinct normal forms n_1 and n_2. We show: If x has two distinct normal forms, x has a direct successor with two distinct normal forms. This is a contradiction to "$\to$ terminates".



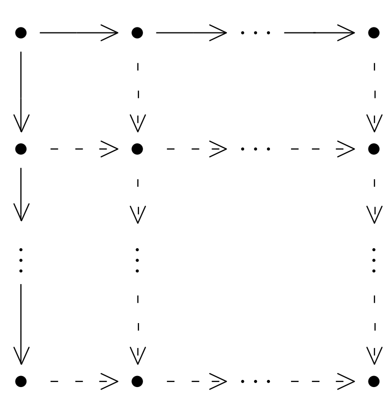
1. $n \neq n_1$: y_1 has two distinct normal forms.
2. $n \neq n_2$: y_2 has two distinct normal forms. □

Example of a locally confluent, but not confluent relation:



Lemma A.2.4. *If $\to$ has the diamond-property, then $\to$ is also confluent.*

Proof: see the following sketch:



□

Lemma A.2.5. *Let $\to$ and $>$ be binary relations with $\to \subseteq > \subseteq \overset{*}{\to}$. Then $\to$ is confluent if $>$ has the diamond-property.*

Proof:

1. Because * is monotone and idempotent, $\rightarrow \subseteq > \subseteq \xrightarrow{*}$ implies $\xrightarrow{*} \subseteq >^* \subseteq (\xrightarrow{*})^* = \xrightarrow{*}$, and thus $\xrightarrow{*} = >^*$.

2. $\quad >$ has the diamond property
   $\Rightarrow \quad >$ is confluent (Lemma A.2.4)
   $\Leftrightarrow \quad >^*$ has the diamond property
   $\Leftrightarrow \quad \xrightarrow{*}$ has the diamond property
   $\Leftrightarrow \quad \rightarrow$ is confluent. □

Definition A.2.6. A relation $\rightarrow \subseteq A \times A$ has the **Church-Rosser property** if

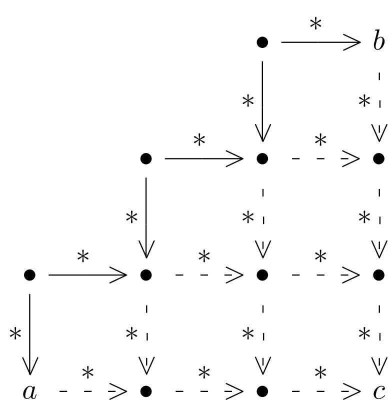
$$a \overset{*}{\leftrightarrow} b \;\Leftrightarrow\; \exists c.\; a \xrightarrow{*} c \xleftarrow{*} b$$

Theorem A.2.7. *A relation $\rightarrow$ is confluent iff it has the Church-Rosser property.*

Proof:

$\Leftarrow$: obvious

$\Rightarrow$: By induction on the number of "peaks" in $a \overset{*}{\leftrightarrow} b$. Informally:



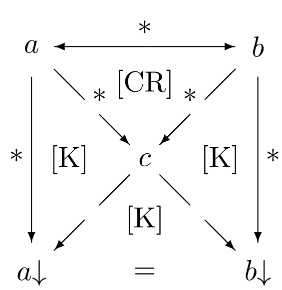
Corollary A.2.8. *If $\rightarrow$ is confluent and if a and b have the normal form $a\downarrow$ and $b\downarrow$, then the following holds:*

$$a \overset{*}{\leftrightarrow} b \quad \Leftrightarrow \quad a\downarrow = b\downarrow$$

Proof:

$\Leftarrow$: obvious

$\Rightarrow$:



[K]: confluence of $\rightarrow$
[CR]: The Church-Rosser property of $\rightarrow$

A.3 Commuting relations

Definition A.3.1. Let $\rightarrow_1$ and $\rightarrow_2$ be arbitrary relations. $\rightarrow_1$ and $\rightarrow_2$ **commute** if for all s, t_1, t_2 the following holds:

$$(s \rightarrow_1 t_1 \wedge s \rightarrow_2 t_2) \Rightarrow \exists u. (t_1 \rightarrow_2 u \wedge t_2 \rightarrow_1 u)$$

$$\begin{tikzcd}
\bullet \arrow[r, "1"'] \arrow[d, "2"] & \bullet \arrow[d, dashed, "2"] \\
\bullet \arrow[r, dashed, "1"'] & \bullet
\end{tikzcd}$$

Lemma A.3.2 (Hindley/Rosen)**.** *If $\rightarrow_1$ and $\rightarrow_2$ are confluent, and if $\overset{*}{\rightarrow}_1$ and $\overset{*}{\rightarrow}_2$ commute, then $\rightarrow_{12} := \rightarrow_1 \cup \rightarrow_2$ is also confluent.*

Proof:

$$\begin{tikzcd}
\bullet \arrow[r, "*", "1"'] \arrow[d, "*"', "2"] \arrow[dr, phantom, "\text{[Km]}"] & \bullet \arrow[r, "*", "2"'] \arrow[d, dashed, "*"', "2"] \arrow[dr, phantom, "\text{[Kf]}"] & \bullet \arrow[r, "*", "1"'] \arrow[d, dashed, "*"', "2"] \arrow[dr, phantom, "\text{[Km]}"] & \bullet \arrow[d, dashed, "*"', "2"] \\
\bullet \arrow[r, dashed, "*", "1"'] \arrow[d, "*"', "1"] \arrow[dr, phantom, "\text{[Kf]}"] & \bullet \arrow[r, dashed, "*", "2"'] \arrow[d, dashed, "*"', "1"] \arrow[dr, phantom, "\text{[Km]}"] & \bullet \arrow[r, dashed, "*", "1"'] \arrow[d, dashed, "*"', "1"] \arrow[dr, phantom, "\text{[Kf]}"] & \bullet \arrow[d, dashed, "*"', "1"] \\
\bullet \arrow[r, dashed, "*", "1"'] & \bullet \arrow[r, dashed, "*", "2"'] & \bullet \arrow[r, dashed, "*", "1"'] & \bullet
\end{tikzcd}$$

[Kf]: $\rightarrow_1$ or rather $\rightarrow_2$ is confluent.
[Km]: $\rightarrow_1$ and $\rightarrow_2$ commute. $\square$

Lemma A.3.3.

$$\begin{tikzcd}
\bullet \arrow[r, "1"'] \arrow[d, "2"] & \bullet \arrow[d, dashed, "*"', "2"] \\
\bullet \arrow[r, dashed, "=", "1"'] & \bullet
\end{tikzcd} \quad \Rightarrow \quad \overset{*}{\rightarrow}_1 \textit{ and } \overset{*}{\rightarrow}_2 \textit{ commute.}$$

Proof:

$$\begin{tikzcd}
s \arrow[r, "1"'] \arrow[d, "2"] & \bullet \arrow[r, "1"'] \arrow[d, "*"', "2"] & \cdots \arrow[r, "1"'] & t \arrow[ddd, "*"', "2"] \\
\bullet \arrow[r, "=", "1"'] \arrow[d, "2"] & \bullet \arrow[d, "*"', "2"] & & \\
\bullet \arrow[r, "=", "1"'] \arrow[d, "2"] & \bullet \arrow[d, "*"', "2"] & & \\
u \arrow[r, "=", "1"'] & \bullet \arrow[r, "=", "1"'] & \cdots \arrow[r, "=", "1"'] & \bullet
\end{tikzcd}$$

Formally: use an induction first on the length of $s \rightarrow_1^* t$, and then use an induction on the length of $s \rightarrow_2^* u$. $\square$

Bibliography

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North-Holland, 2nd edition, 1984.

[GLT90] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[Han04] Chris Hankin. *An Introduction to Lambda Calculi for Computer Scientists.* King's College Publications, 2004.

[HS08] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators. An Introduction.* Cambridge University Press, 2008.

[Loa98] Ralph Loader. Notes on simply typed lambda calculus. Technical Report ECS-LFCS-98-381, Department of Computer Science, University of Edinburgh, 1998.