Lambda Calculus

Prof. Tobias Nipkow

August 2, 2012

Contents

Chapter 1

Untyped Lambda Calculus

1.1 Syntax

1.1.1 Terms

Definition 1.1.1 In lambda calculus the set of **terms** are 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:

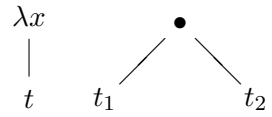
$$\begin{array}{rll} ((\lambda x.((f\ x)x))5) & \rightarrow_\beta & ((f\ 5)5) \\ ((\lambda x.x)(\lambda x.x)) & \rightarrow_\beta & (\lambda x.x) \\ (x(\lambda y.y)) & \text{cannot be reduced} & \end{array}$$

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

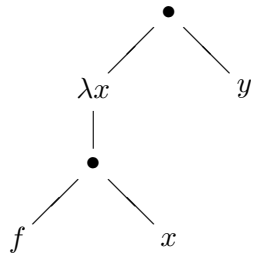
Notation:

- Variables are listed after λ: $\lambda x_1 \ldots x_n.s \equiv \lambda x_1.\ldots.\lambda x_n.s$
- Application associates to the left: $(t_1 \ldots t_n) \equiv (((t_1\ t_2)t_3) \ldots t_n)$
- λ binds to the right as far as possible.
  Example: $\lambda x.x\ x \equiv \lambda x.(x\ x) \not\equiv (\lambda x.x)\ x$
- Outermost parentheses are omitted: $t_1 \ldots t_n \equiv (t_1 \ldots t_n)$

Terms as trees:

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

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, $r\ s\ (t\ u) \equiv (r\ s)\ (t\ u)$

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: $\lambda(x, y).x + y$
Not permitted by lambda calculus syntax!

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

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

Evaluation in lambda-calculus:

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

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	$\cdot$	$\cdot$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

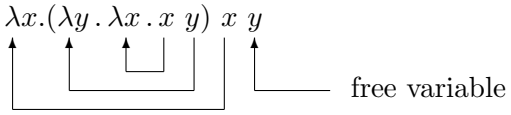
$g'\ 5$ corresponds to the unary function that is given by row 5 of the table.

In set theory: $(A \times B) \to C \quad \cong \quad A \to (B \to C)$
("'$\cong$"': 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:



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:

FV: term $\to$ set of variables

$$\begin{array}{ll} FV(c) & = \emptyset \\ FV(x) & = \{x\} \\ FV(s\ t) & = FV(s) \cup FV(t) \\ FV(\lambda x.t) & = FV(t) \setminus \{x\} \end{array}$$

Definition 1.1.3 A term 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:

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

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 $z \notin FV(t) \cup FV(s)$. 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

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

Remark: These equations hold only up to renaming of bound variables. For example, take equation 1 with $s = \lambda y.y$: $(\lambda y.y)[x/x] = (\lambda z.y[z/y][x/x]) = (\lambda z.z) \neq (\lambda y.y)$. We will identify terms like $\lambda y.y$ and $\lambda z.z$ 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:

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

Definition 1.1.6

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

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:

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

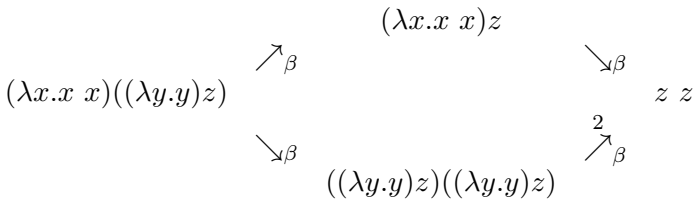


Figure 1.1: $\rightarrow_\beta$ is confluent?

Convention:

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

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

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

   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 $\rightarrow_\beta$.

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

β-reduction is

- nondeterministic: a term may have more than one β-reduct. Example: see Fig. 1.1.
- confluent: see below
- non-terminating. Example: $\Omega := (\lambda x.x\ x)(\lambda x.x\ x) \rightarrow_\beta \Omega$.

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. $s \to_\beta s' \quad \Rightarrow \quad (t\ s) \to_\beta (t\ s')$
4. $s \to_\beta s' \quad \Rightarrow \quad \lambda x.s \to_\beta \lambda x.s'$

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

Lemma 1.2.3 $t \to_\beta^* t' \ \Rightarrow\ s[t/x] \to_\beta^* s[t'/x]$

Proof: by induction on s:

1. $s = x$: obvious
2. $s = y \neq x$: $s[t/x] = y \to_\beta^* y = s[t'/x]$
3. $s = c$: as in 2.
4. $s = (s_1\ s_2)$:
   $$(s_1\ s_2)[t/x] \quad = \quad (s_1[t/x])\ (s_2[t/x]) \quad \to_\beta^* \quad (s_1[t'/x])\ (s_2[t/x]) \quad \to_\beta^*$$
   $$\to_\beta^* \quad (s_1[t'/x])\ (s_2[t'/x]) \quad = \quad (s_1\ s_2)[t'/x] \quad = \quad s[t'/x]$$
   (using the induction hypothesis $s_i[t/x] \to_\beta^* s_i[t'/x]$, $i = 1, 2$, as well as transitivity of $\to_\beta^*$)
5. $s = \lambda y.r$: $s[t/x] \ = \ \lambda y.(r[t/x]) \ \to_\beta^* \ \lambda y.(r[t'/x]) \ = \ (\lambda y.r)[t'/x] \ = \ s[t'/x]$
   (using the induction hypothesis $r[t/x] \to_\beta^* r[t'/x]$) □

Lemma 1.2.4 *The four rules in Definition 1.2.2 are valid with $\to_\beta^*$ in place of $\to_\beta$.*

Lemma 1.2.5 $s \to_\beta s' \ \Rightarrow \ s[t/x] \to_\beta s'[t/x]$

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

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

Corollary 1.2.6 $s \to_\beta^n s' \ \Rightarrow \ s[t/x] \to_\beta^n s'[t/x]$

Proof: by induction on n □

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

Proof: $s[t/x] \overset{*}{\to} s'[t/x] \overset{*}{\to} s'[t'/x]$

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

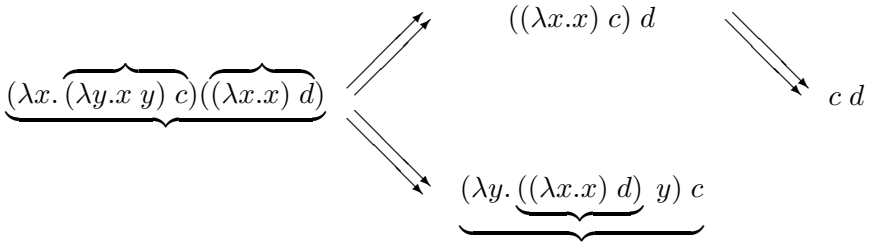
Exercise 1.2.8 *Show $s \to_\beta t \ \Rightarrow \ FV(s) \supseteq FV(t)$. Why does $FV(s) = FV(t)$ not hold?*

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:



$(\lambda y.((\lambda x.x)d)y)c \rightrightarrows c\ d$ 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: $u \stackrel{*}{\rightarrow} u', v \stackrel{*}{\rightarrow} v'$
> $s = (\lambda x.u)v \rightarrow_\beta^* (\lambda x.u')v \rightarrow_\beta^* (\lambda x.u')v' \rightarrow_\beta u'[v'/x]$
> Remaining cases: left as exercise □

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

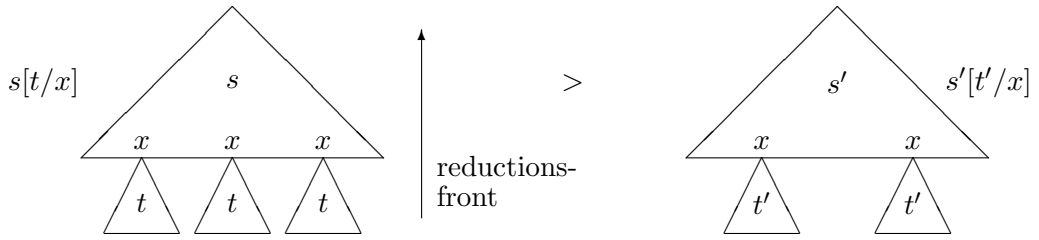
The next lemma follows directly from the analysis of applicable rules:

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 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 $i = 1, 2$)
   $\Rightarrow \exists u'.\, t_i' > u' \quad (i = 1, 2) \qquad$ (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 *$\to_\beta$ is confluent.*

1.3 η-reduction

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

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

$$(\lambda x.(t\ x))u \rightarrow_\beta t\ u$$

if $x \notin FV(t)$.

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

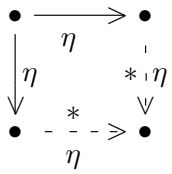
Definition 1.3.1 $C[\lambda x.(t\ x)]\ \rightarrow_\eta\ C[t] \quad$ if $x \notin FV(t)$.

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

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

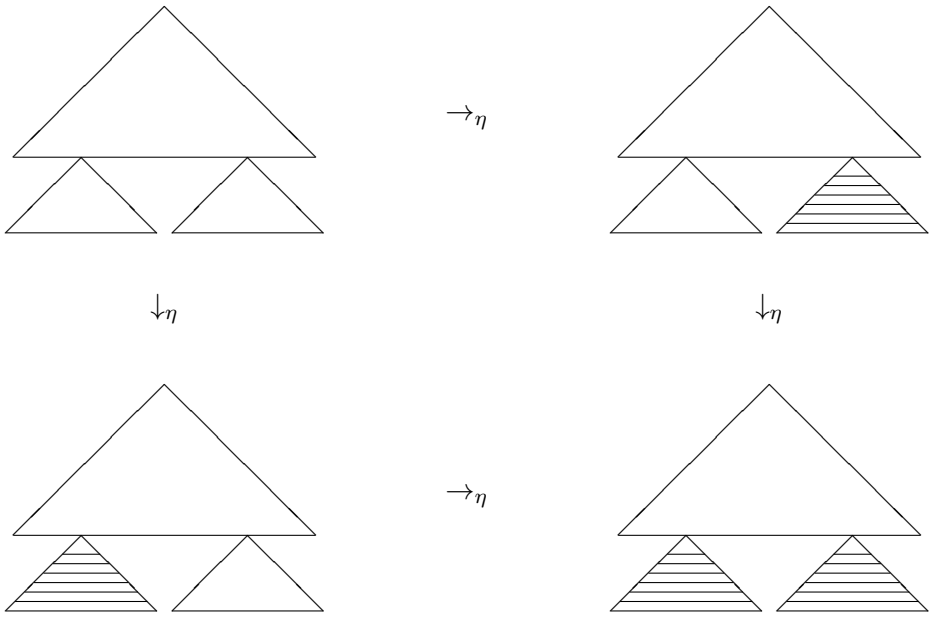
Fact 1.3.3 $s \rightarrow_\eta t \quad \Rightarrow \quad FV(s) = FV(t)$

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



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 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.*

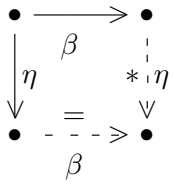
Proof: $\to_\eta$ terminates and is locally confluent.

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

Remark:

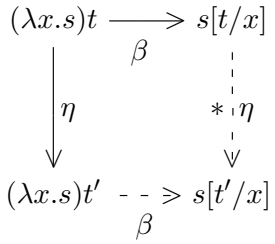
$\to_\eta$ does not have the diamond-property. But one can prove that $\overset{=}{\to}_\eta$ has the diamond-property by slightly modifiying Lemma 1.3.3.

Lemma 1.3.6

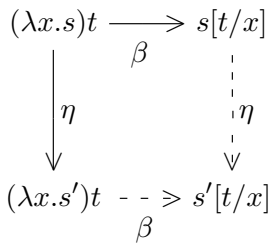


Proof: by case distinction on the relative position of redexes.

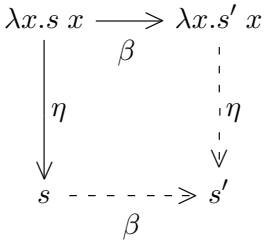
1. In separate subtrees: obvious
2. η-redex far below β-redex:
   (a) $t \to_\eta t'$:



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

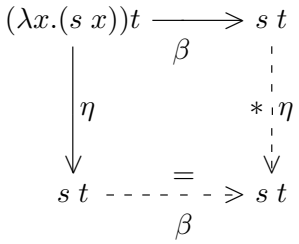


3. β-redex ($s \to_\beta s'$) far below the η-redex:

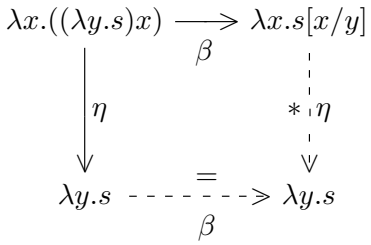


   with help of exercise 1.2.8.

4. β-redex ($s \to_\beta s'$) directly below the η-redex (i.e. overlapped):



5. β-redex directly below the η-redex:



   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, $\overset{*}{\to}_\beta$ and $\overset{*}{\to}_\eta$ 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 $\to_\beta$ is confluent. Undecidability follows from the fact that λ-terms are programs and program equivalences are undecidable. □

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:

$\eta \Rightarrow$ 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 $x \notin FV(s)$: $\forall u.(\lambda x.(s\ x))u =_\beta s\ u \ \Rightarrow\ \lambda x.(s\ x) = s$ □

Definition 1.4.4

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

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.*

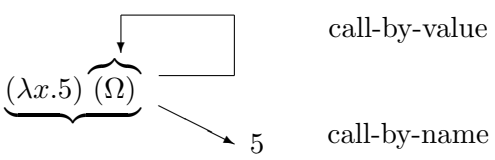
Since $\to_\eta$ is terminating and confluent, the following corollary holds.

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

1.5 Reduction strategies

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.*

Example ($\Omega := (\lambda x.x\ x)(\lambda x.x\ x)$):

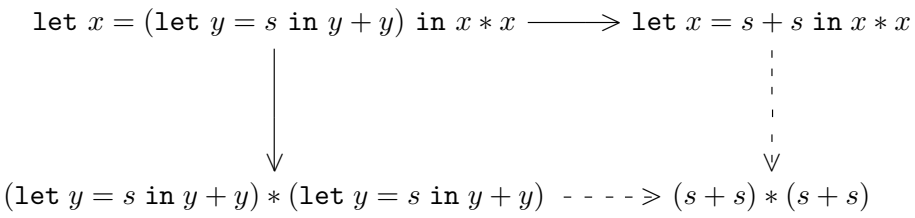


1.6 Labeled terms

Motivation: `let`-expression

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

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



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:

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

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

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

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

but

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

If $s \rightarrow_{\underline{\beta}} t$, 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. $s \in \underline{\mathcal{T}} \ \wedge\ s \rightarrow_{\underline{\beta}} t \ \Rightarrow\ t \in \underline{\mathcal{T}}$

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 $\rightarrow_{\underline{\beta}}$ 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:

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

   $\Rightarrow$ 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 *$\to_{\underline{\beta}}$ is confluent.*

Proof: $\to_{\underline{\beta}}$ is locally confluent. (Use termination and Newmanns Lemma.) □

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

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

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

1.7 Lambda calculus as a programming language

1.7.1 Data types

- `bool`:

  `true`, `false`, `if` with `if true` $x\ y \to^*_\beta x$
  and `if false` $x\ y \to^*_\beta y$

is realized by

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

- Pairs:

  $\texttt{fst}$, $\texttt{snd}$, $\texttt{pair}$ with $\texttt{fst}(\texttt{pair}\ x\ y) \to_\beta^* x$
  and $\texttt{snd}(\texttt{pair}\ x\ y) \to_\beta^* y$

  is realized by

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

  Example:

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

- $\texttt{nat}$ (Church-Numerals):

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

  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:

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

Exercise 1.7.1

1. *Lists in λ-calculus: Find λ-terms for* $\texttt{nil}$, $\texttt{cons}$, $\texttt{hd}$, $\texttt{tl}$, $\texttt{null}$ *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* `mult` $\underline{m}\ \underline{n} \xrightarrow{*} \underline{m*n}$
   and `expt` *with* `expt` $\underline{m}\ \underline{n} \xrightarrow{*} \underline{m^n}$
3. *Difficult: Find* `pred` *with* `pred` $\underline{m+1} \xrightarrow{*} \underline{m}$ *and* `pred` $\underline{0} \xrightarrow{*} \underline{0}$

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` a fixed point operator, i.e. `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:

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

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

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

Note: even `add` $\underline{1}\ \underline{2} \xrightarrow{*}_\beta \underline{3}$ 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"

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

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

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

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

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

1.7.3 Computable functions on $\mathbb{N}$

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

1. $t\ \underline{m_1}\ \ldots\ \underline{m_n} \to_\beta^* \underline{m}$, 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

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.

   `True`, `False`, $\wedge$, ... are λ-terms

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

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

   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 $\to \{k \mid 0 \le k < m\}$

   Result type depends on the input value

   This approach is known as "type theory".

2.1 Simply typed λ-calculus ($\lambda^{\to}$)

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:

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

Convention: $\to$ associates to the right:

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

Terms:

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

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

2.1.1 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:

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

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

Examples:

- A simple derivation:
$$\frac{\Gamma[x : \tau] \vdash x : \tau}{\Gamma \vdash \lambda x : \tau.x : \tau \to \tau}$$
- 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
$$\dfrac{\dfrac{\dfrac{\tau = \tau_2 \to \tau_1}{\Gamma[x : \tau] \vdash x : \tau_2 \to \tau_1} \quad \dfrac{\tau = \tau_2}{\Gamma[x : \tau] \vdash x : \tau_2}}{\Gamma[x : \tau] \vdash (x\ x) : \tau_1} \qquad \tau' = \tau \to \tau_1}{\Gamma \vdash \lambda x : \tau.(x\ x) : \tau'}$$
$\Rightarrow$ Contradiction: $\neg\exists \tau_1, \tau_2 : \ \tau_2 \to \tau_1 = \tau_2$

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:

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

Definition 2.1.1 t is **type-correct** (with regard to Γ), if there exists τ such that $\Gamma \vdash t : \tau$.

Lemma 2.1.2 *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 2.1.3 *Each subterm of a type-correct term is type-correct.*

This is obvious from the rules.

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 2.1.4 (Subject reduction) $\Gamma \vdash t : \tau \wedge t \rightarrow_\beta t' \ \Rightarrow\ \Gamma \vdash t' : \tau$

This does not hold for β-expansion:

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

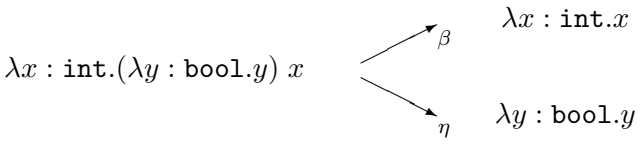
and

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

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

Theorem 2.1.5 $\rightarrow_\beta$ *($\rightarrow_\eta$,$\rightarrow_{\beta\eta}$) over type-correct terms is confluent.*

This does not hold for all raw terms:



Theorem 2.1.6 $\rightarrow_\beta$ *terminates over type-correct terms.*

The proof is discussed in Section 2.2. A vague intuition is that the type system forbids self-application and thus recursion. This has the following positive consequence:

Corollary 2.1.7 $=_\beta$ *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 2.1.6 is the following:

Corollary 2.1.8 *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 2.1.9 *Let Y_τ be a family of constants of type $(\tau \to \tau) \to \tau$ that reduce like fixed-point combinators: $Y_\tau\, t \;\to\; t\,(Y_\tau\; t)$. Then every computable function can be represented as a closed type-correct $\lambda^{\to}$-term which contains as its only constants the Y_τ.*

Proof sketch:

1. Values of basic types (booleans, natural numbers, etc) are representable by type-correct $\lambda^{\to}$-terms.
2. Recursion with Y_τ

2.2 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 [HS86, GLT90, Han04].

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

Definition 2.2.1 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:

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

Lemma 2.2.2 $t \in T \Rightarrow t{\Downarrow}$

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

(Var) $(x\, r_1 \ldots r_n){\Downarrow}$ follows directly from $r_1{\Downarrow}$, …, $r_n{\Downarrow}$, since x is a variable.

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

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

$$(\lambda x.r)\, s\, s_1 \ldots s_n \to^*_\beta (\lambda x.r')\, s'\, s'_1 \ldots s'_n \to_\beta r'[s'/x]\, s'_1 \ldots s'_n \to_\beta \cdots$$

since r, s (by I.H.) and all s_i terminate. However, $r[s/x]\, s_1 \ldots s_n \to^*_\beta r'[s'/x]\, s'_1 \ldots s'_n$ also holds. This contradicts the termination of $r[s/x]\, s_1 \ldots s_n$. Therefore $(\lambda x.r)\, s\, s_1 \ldots s_n$ cannot diverge. □

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 2.2.3 *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 $t = x\,r_1 \ldots r_n$ and all $r_i \in T$, then $t\,r = x\,r_1 \ldots r_n\,r \in T$ 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 $t = (\lambda x.r)\,s\,s_1 \ldots s_n$ and $r[s/x]\,s_1 \ldots s_n \in T$ and $s \in T$, then by I.H. $r[s/x]\,s_1 \ldots s_n\,r \in T$ holds. Since $s \in T$, $t\,r = (\lambda x.r)\,s\,s_1 \ldots s_n\,r \in T$ follows with (β). □

Lemma 2.2.4 *Let $\tau = \tau_1 \to \cdots \to \tau_k \to \tau'$, 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 $s = y\,s_1 \ldots s_n$ and all $s_i \in T$, then $s_i[r/x] \in T$ holds by I.H., $i = 1, \ldots, n$. If $x \neq y$, then $s[r/x] = y(s_1[r/x]) \ldots (s_n[r/x]) \in T$ by (Var). If $x = y$, then $y : \tau$ holds, and therefore $s_i : \tau_i$, and $s_i[r/x] : \tau_i$, $i = 1, \ldots, n$ as well. Since all τ_i are applicative, $s[r/x] = r(s_1[r/x]) \ldots (s_n[r/x]) \in T$ 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 $s = (\lambda y.u)\,s_0\,s_1 \ldots s_n$ by $u[s_0/y]\,s_1 \ldots s_n \in T$ and $s_0 \in T$, then $s[r/x] = (\lambda y.(u[r/x]))$ $(s_0[r/x]) \ldots (s_n[r/x]) \in T$ follows by (β) since $u[r/x][s_0[r/x]/y](s_1[r/x]) \ldots (s_n[r/x])$
$= (u[s_0/y]\,s_1 \ldots s_n)[r/x] \in T$ and $s_0[r/x] \in T$ by I.H. □

Exercise 2.2.5 *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 2.2.6 *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 2.2.5 from the I.H. $r \in T$ and $s \in T$. □

Theorem 2.1.6 is just a corollary of Theorem 2.2.6 and Lemma 2.2.2.

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

Types: τ ::= $\texttt{bool} \mid \texttt{int} \mid \ldots$ basic types
$\mid \alpha \mid \beta \mid \gamma \mid \ldots$ type variables
$\mid \tau_1 \to \tau_2$

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):

$$\begin{aligned} \lambda x.x : &\quad \alpha \to \alpha \\ \lambda x.x : &\quad \texttt{int} \to \texttt{int} \end{aligned}$$

Definition 2.3.1 $\tau_1 \gtrsim \tau_2 \ :\Leftrightarrow\ \exists$ 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:

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

Every type-correct term has a most general type:

Theorem 2.3.2 $\Gamma \vdash t : \tau \quad \Rightarrow \quad \exists \sigma.\ \Gamma \vdash t : \sigma\ \wedge\ \forall \tau'. \Gamma \vdash t : \tau'\ \Rightarrow\ \tau' \gtrsim \sigma$

Proof idea: Consider rules as a Prolog program that compute the type from the term. Given the term, the rules are deterministic, i.e. at most one rule is applicable at any point. Hence there is at most one type. Because rule application uses unification, it computes the most general type (via the most general unifier).

Example, using Roman instead of Greek letters as type variables:

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

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

2.4 `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 `let`s is uniquely defined because of termination and confluence of $\to_{\underline{\beta}}$. We will now examine type inference in the presence of `let`.

Example:

$$\texttt{let} \quad \underbrace{f = \lambda x.x}_{f : \forall \alpha.\underbrace{\alpha \to \alpha}_{\tau}} \quad \texttt{in} \quad \text{pair} \quad \underbrace{(f\ 0)}_{f : \tau[\texttt{int}/\alpha]} \quad \underbrace{(f\ \text{true})}_{f : \tau[\texttt{bool}/\alpha]}$$

Note

- $\forall$-quantified type variables can be replaced by arbitrary types.
- Although $(\lambda f.\text{pair}\ (f\ 0)\ (f\ \text{true}))\ (\lambda x.x)$ 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 2.3 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 $\forall\alpha_1 \ldots \forall\alpha_n.\tau$, compactly written $\forall\alpha_1 \ldots \alpha_n.\tau$.

Example of type schemas are α, `int`, $\forall\alpha.\alpha \to \alpha$ and $\forall\alpha\beta.\alpha \to \beta$. Note that $(\forall\alpha.\alpha \to \alpha) \to$ `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 $[x_1 : \sigma_1, \ldots, x_n : \sigma_n]$:

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

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

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

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

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

Then there are the quantifier rules:

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

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

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

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

Logic: $\quad x = 0 \vdash x = 0 \ \not\Rightarrow \ x = 0 \vdash \forall x.x = 0$

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

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

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

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):

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

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma[x : \forall\alpha_1 \ldots \alpha_n.\tau] \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : \tau_2}\ \text{(Let')} \qquad \{\alpha_1, \ldots, \alpha_n\} = FV(\tau) \setminus FV(\Gamma)$$

(Var) and (Let) are replaced by (Var') and (Let'), respectively. (App) and (Abs) remain unchanged. (∀Intro) and (∀Elim) disappear: The resulting system has four syntax-directed rules. Note: Type schemas occur only in Γ.

Example:

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

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

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

Proof of the equivalence of two systems: Each derivation tree with explicit quantifier rules can be transformed in such a way that (∀Elim) is found only under the (Var)-rules and (∀Intro) only in the left premise of the (`let`)-rule.

Complexity of type inference:

- without `let`: linear
- with `let`: DEXPTIME-complete (Types can grow exponentially with the size of the terms.)

Example:

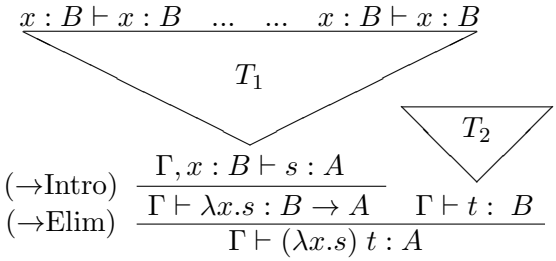
$\texttt{let}\ x_0 = \lambda y.\lambda z.z\ y\ y$
$\texttt{in let}\ x_1 = \lambda y.x_0\ (x_0\ y)$
$\quad\texttt{in}\ \ldots$
$\qquad\ddots$
$\qquad\quad\texttt{let}\ x_{n+1} = \lambda y.x_n\ (x_n\ y)$
$\qquad\quad\texttt{in}\ x_{n+1}\ (\lambda z.z)$

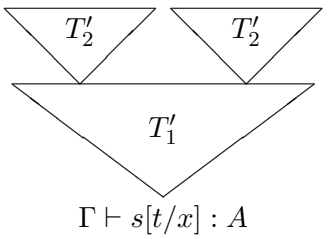
Chapter 3

The Curry-Howard Isomorphismus

typed λ-calculus ($\lambda^{\to}$)	constructive logic (intuitionistic propositional logic)
Types: $\tau ::= \alpha \mid \beta \mid \gamma \mid \ldots \mid \tau \to \tau$	Formulas: $A ::= \underbrace{P \mid Q \mid R \mid \ldots}_{\text{propositional variable}} \mid A \to A$
$\Gamma \vdash t : \tau$	$\Gamma \vdash A$ (Γ: 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.

Proofs where the first premise of $\to$Elim is proved by $\to$Intro 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:

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

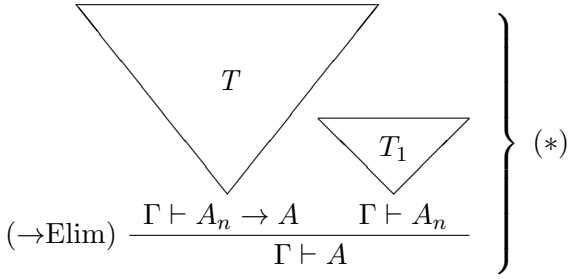
Definition 3.0.1 A proof is in **normal form** if the corresponding λ-term is in normal form.

A proof is in normal form if and only if no part of the proof has the following form.

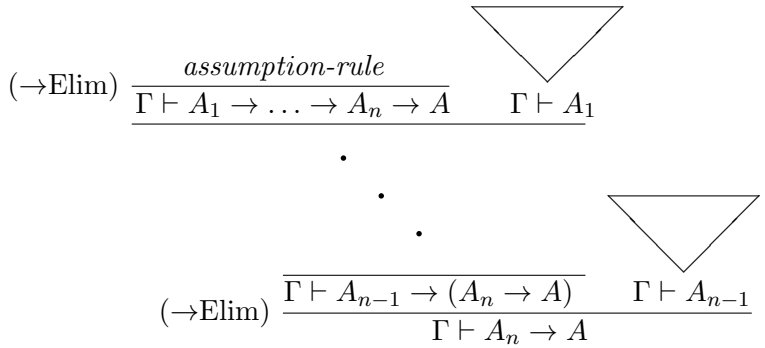
$$(\to \text{Elim})\dfrac{(\to \text{Intro})\dfrac{\dots}{\dots} \qquad \dots}{\dots}$$

The following lemma follows directly.

Lemma 3.0.2 *A proof in normal form, which ends with* ($\to$Elim)*, has to have the following form.:*



where the subtree T has the following form:



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

Theorem 3.0.3 *In a proof of* $\Gamma \vdash A$ *in normal form, only subformulas of* Γ *and* A *occur.*

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

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

2.

$$(\to\text{Intro})\ \frac{\begin{array}{c}T\\ \Gamma, A_1 \vdash A_2\end{array}}{\Gamma \vdash A_1 \to A_2}$$

Induction hypothesis: only subformulas of Γ, A_1 and A_2 occur in T.

Hence the assertion follows immediately.

3. See $(*)$ above.

Because of assumption-rule: $A_1 \to A_2 \to \ldots \to A_n \to A \in \Gamma$.

Ind. hyp. for T_1: in T_1 only subformulas of Γ, $A_n \Rightarrow$ in T_1 only subformulas of Γ.

Ind. hyp. for T: in T only subformulas of Γ, $A_n \to A \ \Rightarrow$ in T only subformulas of Γ.

Therefore, in the whole tree only subformulas of Γ. □

Theorem 3.0.4 $\Gamma \vdash A$ *is decidable.*

The proof is the following algorithm:

Finite search for proof tree in normal form (always exists, since $\to_\beta$ terminates for type-correct terms) by building up from the root to the leaves. As long as the tree is complete, choose unproven leaf:
If $\Gamma \vdash A$ with $A \in \Gamma$, then proof by assumption rule.
Otherwise cycle test: does the leaf already occur on the path to the root?
If yes: backtrack to and modify most recent choice in this subtree.
Otherwise use (→Intro) (premise is uniquely determined) or

$$\frac{\Gamma \vdash A_n \to A \qquad \Gamma \vdash A_n}{\Gamma \vdash A}\ (\to \text{Elim})$$

so that $A_1 \to \cdots A_n \to A \in \Gamma$ (finite choice).
This algorithm terminates because of the following two reasons. First, the root has only a finite number of subformulas and above the root only these subformulas occur (by construction), i.e. there are only a finite number of $\Gamma' \vdash A'$ which can appear above the root (the context is a set, i.e. no duplicates). Second, cycles are detected. □

Example:

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

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

$$\cfrac{\cfrac{\Gamma \vdash A_n \to P \quad \Gamma \vdash A_n}{\Gamma := (P \to Q) \to P \vdash P}\ (\to \text{Elim})}{\vdash ((P \to Q) \to P) \to P}\ (\to \text{Intro})$$

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 (Elim), because Γ does not contain any formula of the form $\cdots \to (P \to Q)$. Hence:

$$\dfrac{\dfrac{\Gamma, P \vdash B_n \to Q \quad \Gamma, P \vdash B_n}{\Gamma, P \vdash Q} \to (\text{Elim})}{\Gamma \vdash P \to Q} \to (\text{Intro})$$

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 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 3.0.5 *Prove* $\vdash ((((p \to q) \to p) \to p) \to q) \to q$.

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:

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

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

Appendix A

Relational Basics

A.1 Notation

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

Definition A.1.1

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

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

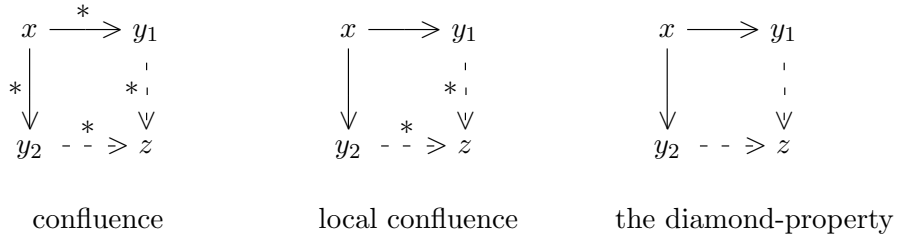
A.2 Confluence

Definition A.2.1 A relation $\to$

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

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

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



confluence local confluence the diamond-property

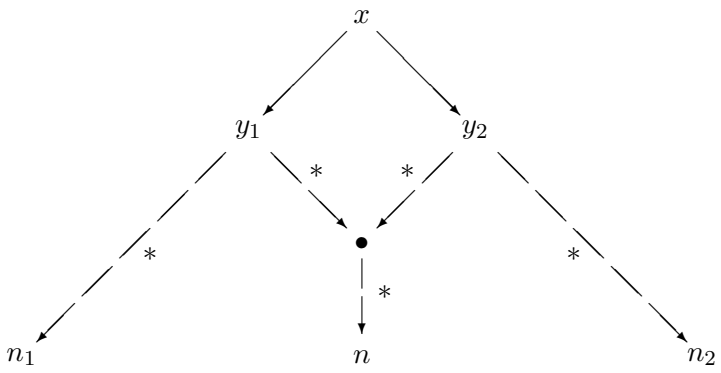
Figure A.1: Sketch of Definition A.2.1

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

Lemma A.2.3 (Newmann's Lemma) *If $\rightarrow$ is locally confluent and terminating, then $\rightarrow$ is also confluent.*

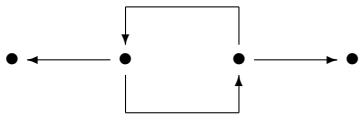
Proof: by contradiction

Assumption: $\rightarrow$ 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 "$\rightarrow$ terminates".



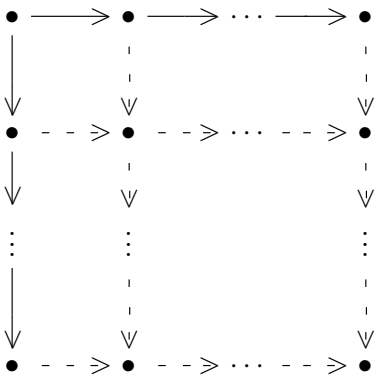
1. $n \neq n_1$: y_1 has two distinct normal forms.
2. $n \neq n_2$: y_2 has two distinct normal forms. □

Example of a locally confluent, but not confluent relation:



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

Proof: see the following sketch:



□

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

Proof:

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

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

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

$$a \overset{*}{\leftrightarrow} b \;\Leftrightarrow\; \exists c.\, a \overset{*}{\to} c \overset{*}{\leftarrow} b$$

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

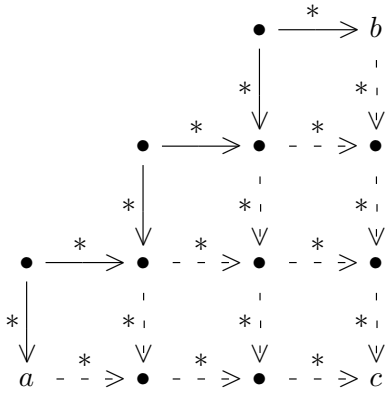
Proof:

"$\Leftarrow$": obvious

"$\Rightarrow$":

1. $a \overset{*}{\to} c \overset{*}{\leftarrow} b \;\Rightarrow\; a \overset{*}{\leftrightarrow} b$

2. $a \leftrightarrow b$:



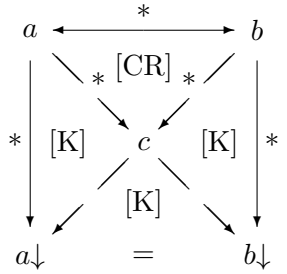
Corollary A.2.8 *If $\to$ is confluent and if a and b have the normal form $a{\downarrow}$ and $b{\downarrow}$, then the following holds:*

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

Proof:

$\Leftarrow$: obvious

$\Rightarrow$:

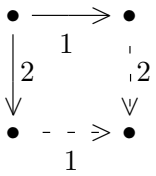


[K]: confluence of $\to$
[CR]: The Church-Rosser property of $\to$

A.3 Commuting relations

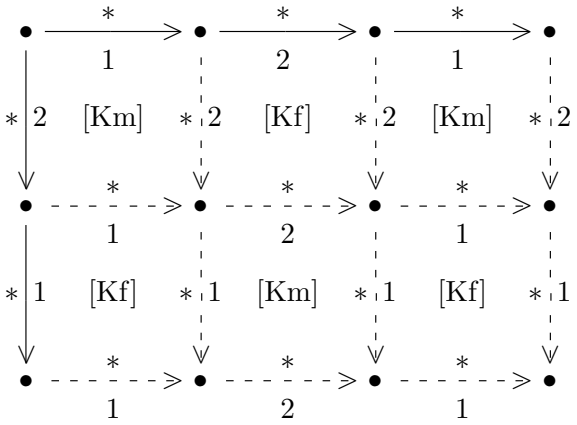
Definition A.3.1 Let $\to_1$ and $\to_2$ be arbitrary relations. $\to_1$ and $\to_2$ **commute** if for all s, t_1, t_2 the following holds:

$$(s \to_1 t_1 \wedge s \to_2 t_2) \Rightarrow \exists u.\, (t_1 \to_2 u \wedge t_2 \to_1 u)$$



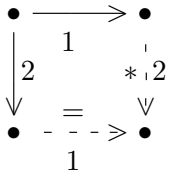
Lemma A.3.2 (Hindley/Rosen) *If $\to_1$ and $\to_2$ are confluent, and if $\overset{*}{\to}_1$ and $\overset{*}{\to}_2$ commute, then $\to_{12} := \to_1 \cup \to_2$ is also confluent.*

Proof:



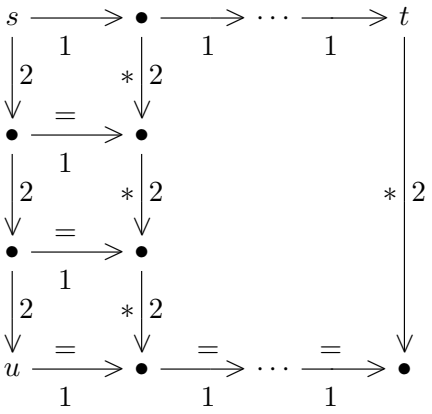
[Kf]: $\to_1$ or rather $\to_2$ is confluent.
[Km]: $\to_1$ and $\to_2$ commute. □

Lemma A.3.3



$\Rightarrow$ $\overset{*}{\to}_1$ *and* $\overset{*}{\to}_2$ *commute.*

Proof:



Formally: use an induction first on the length of $s \to_1^* t$, and then use an induction on the length of $s \to_2^* u$. □

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.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

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