ALP I

λ–Kalkül

WS 2012/2013

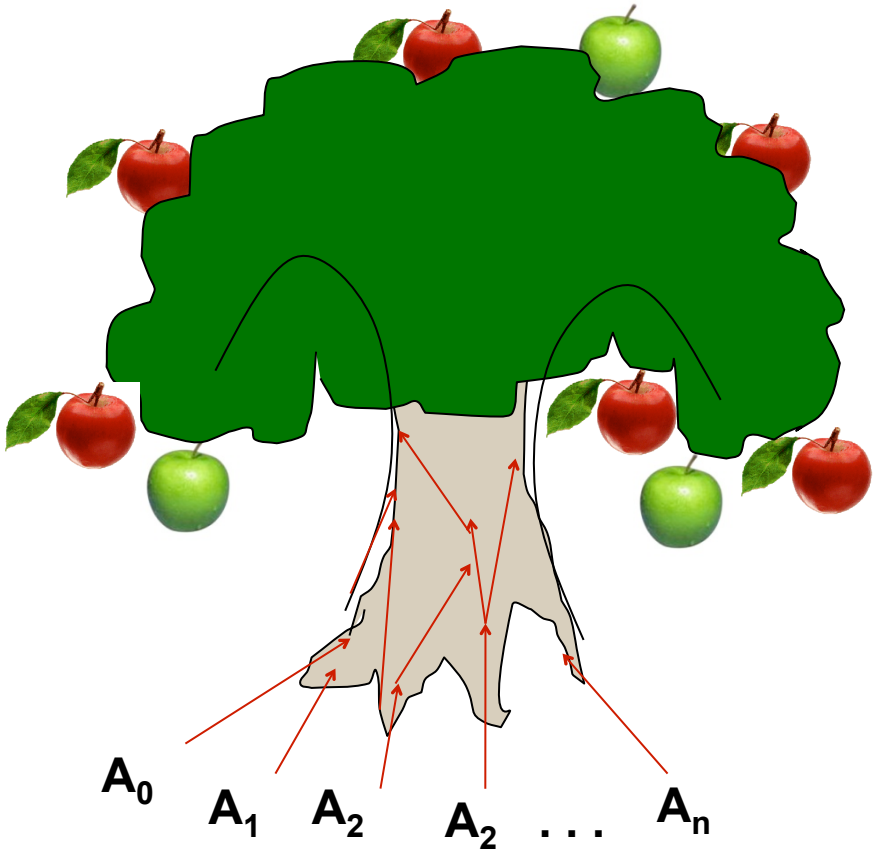
Prof. Dr. Margarita Esponda

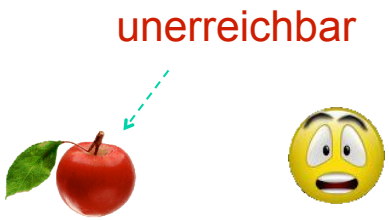
Berechenbarkeit

- inspiriert durch Hilbert's Frage
- im Jahr **1900, Paris**
- Internationaler Mathematikerkongress

Gibt es ein System von Axiomen, aus denen alle Gesetze der Mathematik mechanisch ableitbar sind?

Berechenbarkeitsbegriff

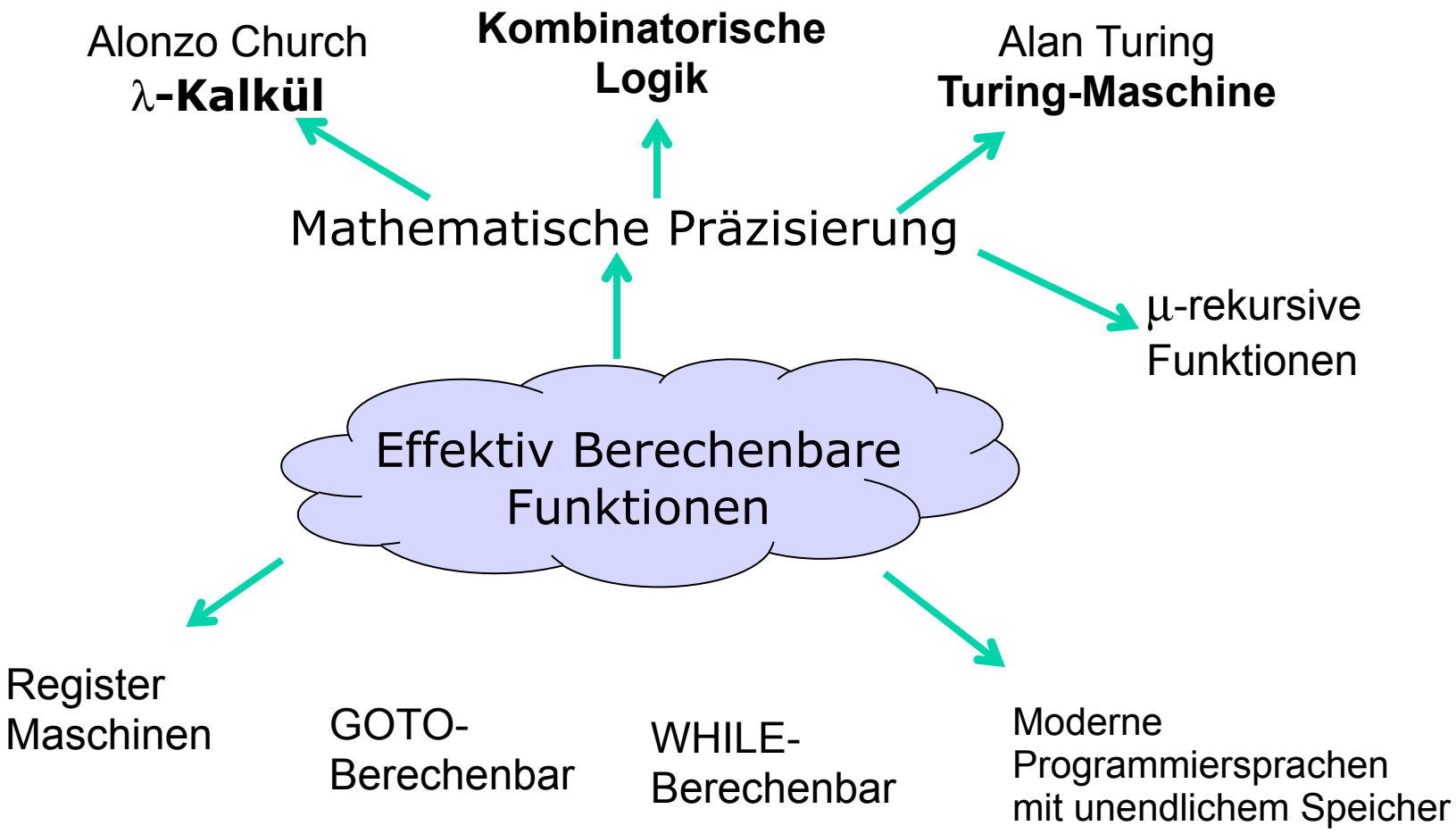




Gödels Unvollständigkeitssätze zeigten, dass Hilberts Programm nicht realisierbar ist.

- Church und Turing.
  Die Formalisierung des Algorithmus-Begriffs

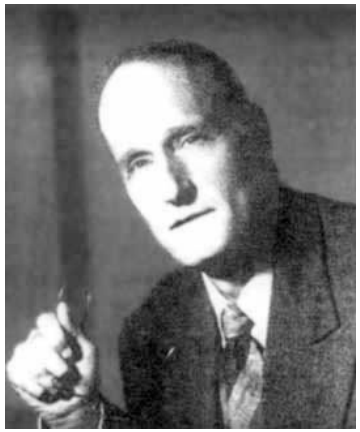
Äquivalenz vieler Berechnungsmodelle

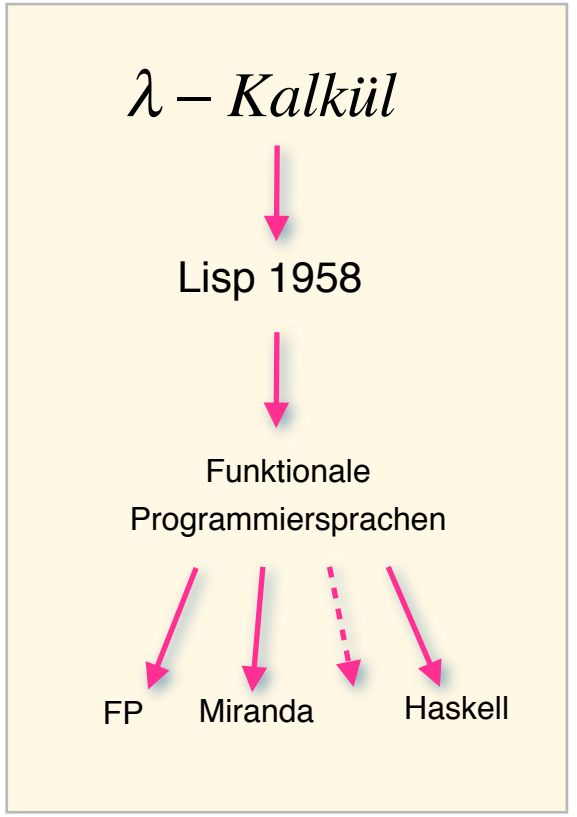


30er Jahre — Erste Programmiersprache

Alonzo Church
1903-1995

Stephen C. Kleene
1909-1994





Lambda-Kalkül

- universelle abstrakte Programmiersprache
- nur die Hardware fehlte

λ–Kalkül

- minimale universelle Programmiersprache
- **Alonzo Church** und **Stephen Kleene**
- **1936**
    - Entscheidungsproblem mit nein beantwortet
    - **Halteproblem** (Turing) nicht lösbar

λ–Kalkül

λ–Kalkül

- minimal, aber es lässt sich alles ausdrücken, was sich mit einer modernen Programmiersprache ausdrücken lässt.
- d.h. alle effektiv berechenbaren Funktionen
  - Funktionen, deren Berechnung sich mit einer endlichen Anzahl von Schritten formulieren lässt (**Algorithmus**)
- Grundlage für funktionale Programmiersprachen

BNF und EBNF

Metasprachen zur Darstellung Kontextfreier Grammatiken

Kontextfreie Grammatiken sind durch ein Tupel G = (T, N, P, S) definiert mit

- T = Menge der Terminalen Symbole
- N = Menge der nicht Terminalen Symbole
- P = Menge der Produktionsregeln
- S = Startsymbol mit $S \in N$

Produktionsregeln haben folgende Form:

$$p \rightarrow w \text{ mit } p \in N \text{ und } w \in N \cup T$$

BNF und EBNF

Metasymbole der EBNF-Metasprache

< … >	Nichtterminal-Symbole werden in spitzen Klammern umrahmt. Die Namen bestehen nur aus kleinen Buchstaben.
|	steht für Alternativen (oder-Verknüpfung) innerhalb von Produktionsregeln
".."	Terminal-Symbole werden zwischen Anführungszeichen geschrieben, wenn diese aus mehreren Buchstaben bestehen, sonnst werden sie einfach direkt geschrieben.
::=	für die Definition von Produktionsregeln verwendet.
[…]	optionale Elemente werden mit eckigen Klammern umrahmt.
{ … }	im geschweiften Klammern werden beliebig wiederholbare Ausdrücke geschrieben.
(…)	für die Gruppierung von Ausdrücken
,	Komma wird machmal als Verkettungs-Operator verwendet.

λ–Kalkül

Ein λ-Ausdruck **E** ist :

1) Ein Variablen-Name

x, z, y, . . ., a, b, . . .

2) Eine Lambda-Abstraktion

λ x. E mit E Lambda-Ausdruck

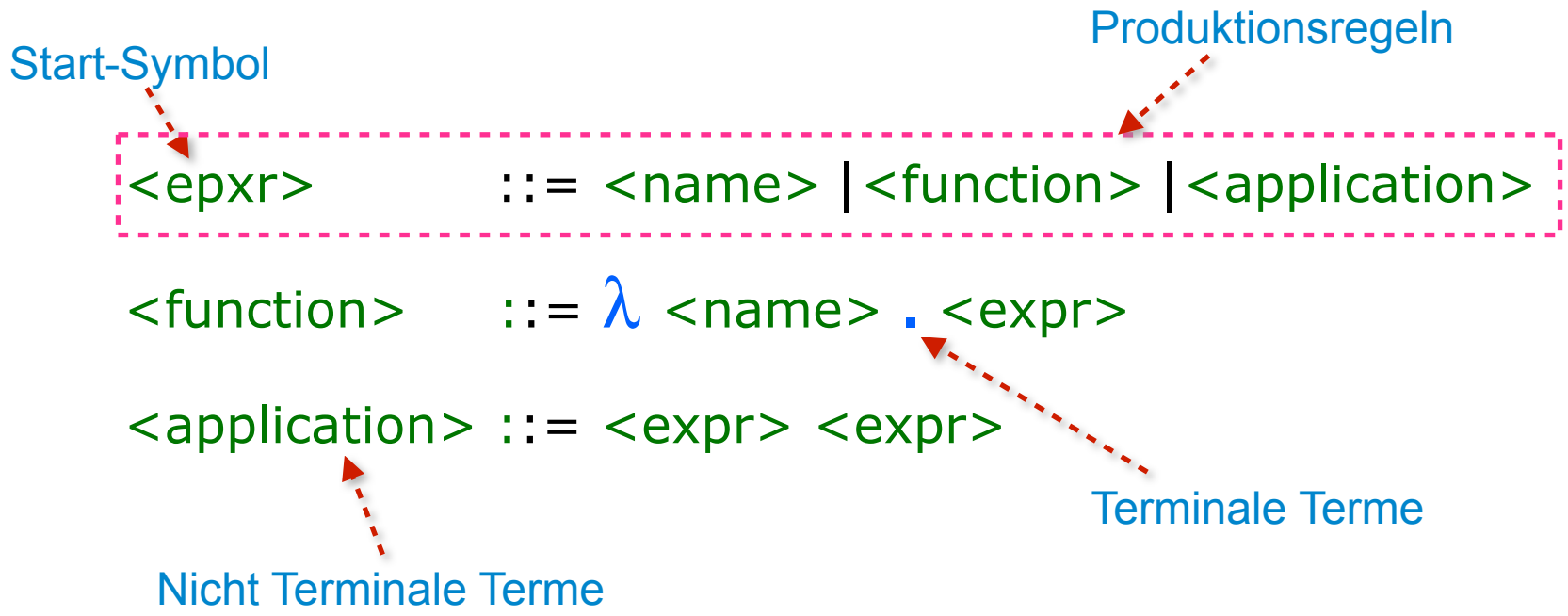
3) Eine Applikation

E_1 E_2 mit E_1 und E_2 Lambda-Ausdrücke

λ–Kalkül

Syntax:

Ein λ-Ausdruck **E** <expr> im **BNF** ist:



λ–Kalkül

Konventionen:

1) Ein Ausdruck in Klammern **(E)** ist äquivalent zu **E**

   $(E) \equiv E$

2) Die Auswertung der Ausdrücke ist linksassoziativ

   $E_1\ E_2\ E_3 \ldots E_n \equiv ((\ldots(E_1\ E_2)\ E_3) \ldots E_n)$

3) Die Funktionsabstraktion ist rechtsassoziativ

   $\lambda x.\ \lambda y.\ \lambda z.\ E \equiv \lambda x.\ (\lambda y.\ (\lambda z.\ E)\ )$

4) Für die Variablennamen werden wir einzelne Buchstaben verwenden.

λ–Kalkül

Wichtige Eigenschaften:

1) Funktionen haben keine Namen.
2) Es gibt keine Datentypen.
3) Kein Unterschied zwischen Funktionsdefinition und Funktionsanwendung.

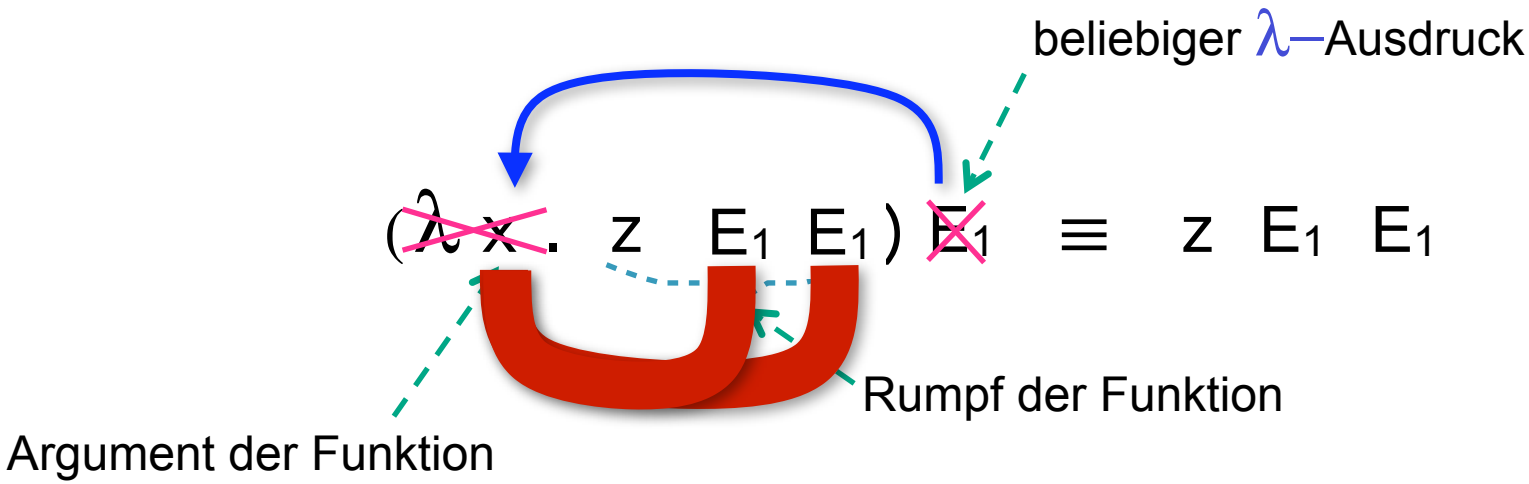
Auswertung von λ–Ausdrücken

Ausdrücke werden durch Applikation reduziert:

Im Ausdruck $(\lambda x . E_1) E_2$ werden alle Vorkommen der Variablennamen **x** in $\mathbf{E_1}$ durch $\mathbf{E_2}$ ersetzt und $\lambda x .$ wird entfernt.

Die Applikation wird so lange wiederholt, bis keine Reduktion mehr möglich wird.

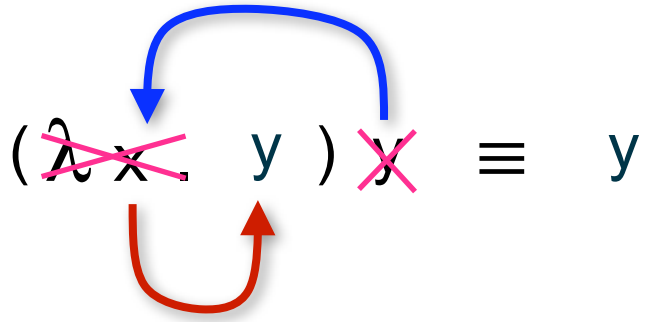
Auswertung von λ–Ausdrücken



λ–Kalkül

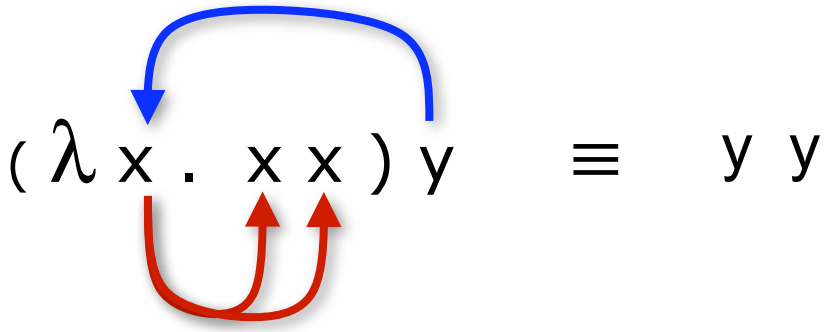
Beispiel:

$\lambda x . x$ definiert die Identitätsfunktion



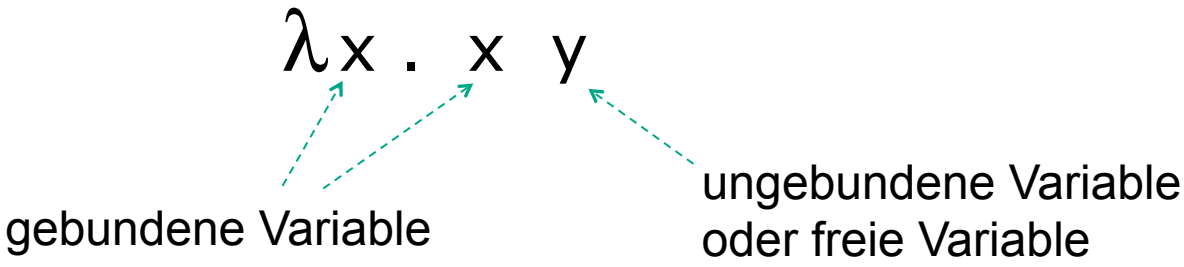
λ–Kalkül

Beispiel:



Gebundene und ungebundene Variablen

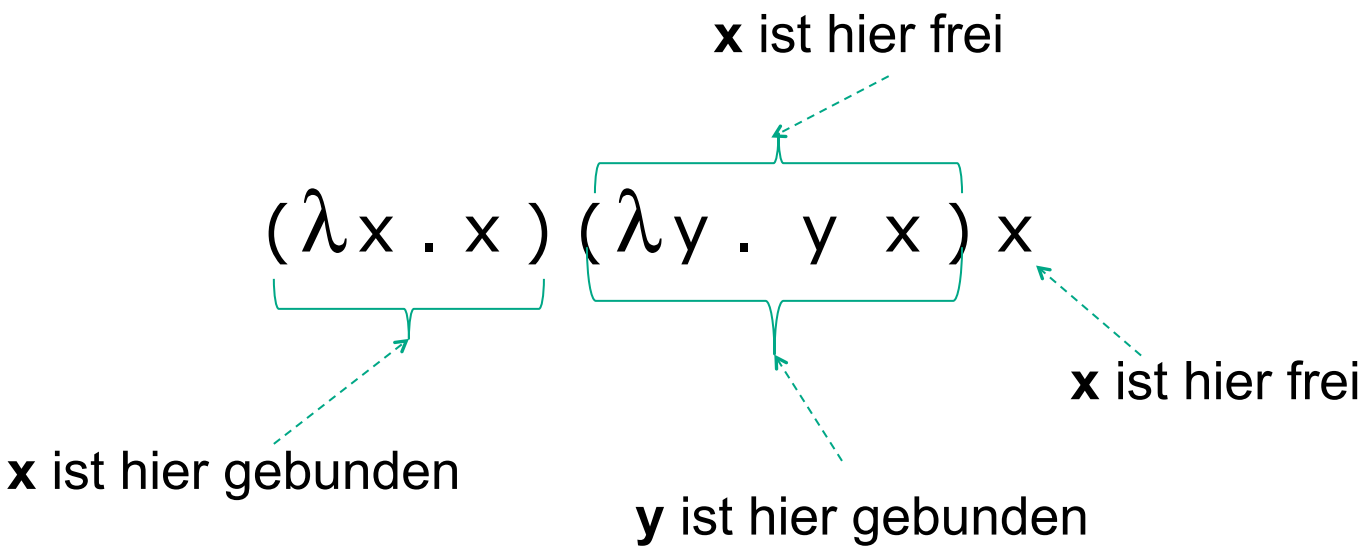
Beispiel:



Eine Variable x ist gebunden, wenn diese sich in dem Rumpf einer Lambda-Abstraktion befindet, dessen Argument gleich x ist (λ x)

Gebundene und ungebundene Variablen

Beispiel:



Gebundene und ungebundene Variablen

Ein Variablenname ist **frei oder ungebunden** innerhalb eines Ausdrucks, wenn folgende Regeln stattfinden:

1) <name> ist frei in <name>

2) $<name_1>$ ist frei in $\lambda <name_2>.$ <expr>, wenn $<name_1> \neq <name_2>$

3) <name> ist frei in $E_1\ E_2$, wenn <name> frei in E_1 oder frei in E_2 ist.

Gebundene und ungebundene Variablen

Ein Variablenname ist **gebunden** innerhalb eines Ausdrucks, wenn folgende Regeln stattfinden:

1) <name$_1$> ist in λ<name$_2$>. <expr> gebunden, wenn <name$_1$> = <name$_2$> ist.

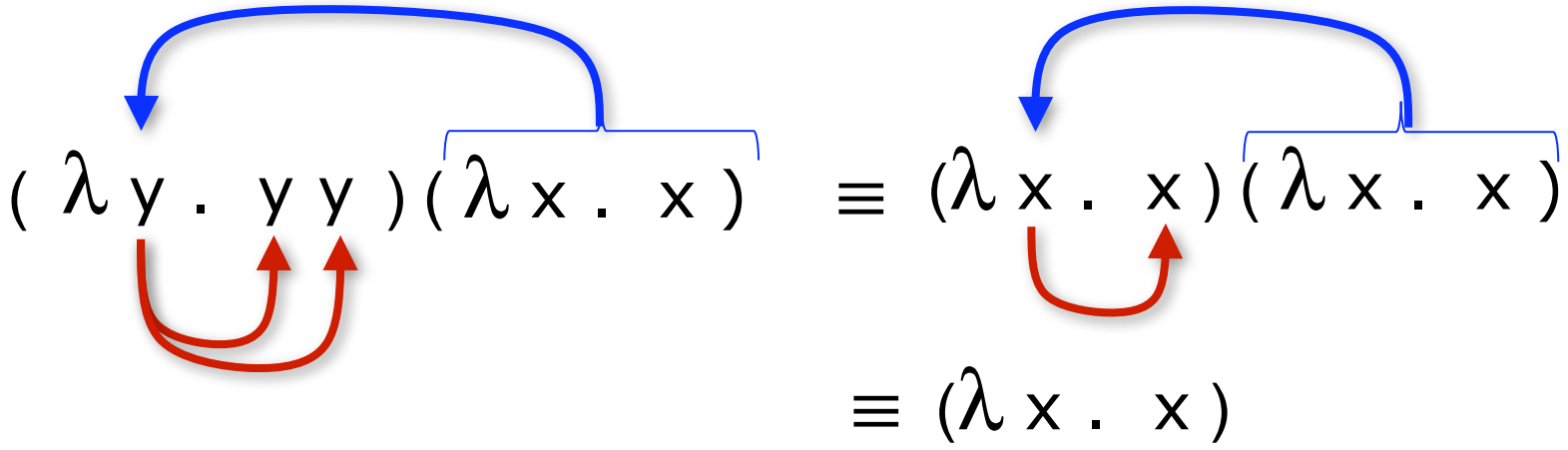
2) <name> ist in E_1 E_2 gebunden, wenn <name> in E_1 oder in E_2 gebunden ist.

Der gleiche Variablenname kann gebunden und ungebunden innerhalb eines Ausdrucks sein!!

Naming-Problem

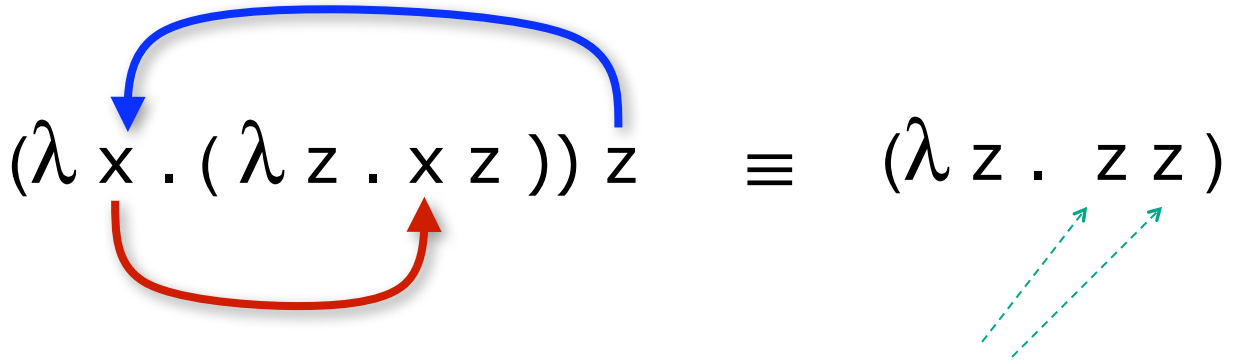
Weil Lambda-Abstraktionen keine Namen haben, müssen die Ausdrücke während der Auswertung vollständig kopiert werden.

Beispiel:



Naming-Problem

Beispiel:



Falsch!

Die **z** Namen waren ursprünglich für völlig unterschiedliche Variablen gedacht.

Variablen müssen vor der Ersetzung umbenannt werden

Folgende Ausdrücke sind äquivalent:

$$\lambda x . x z \equiv \lambda a . a z \equiv \lambda \square . \square z$$

Vor der Ersetzung werden Variablen umbenannt

$$(\lambda x.(\lambda z.xz))z \equiv (\lambda x.(\lambda a.xa))\ z$$

$$\equiv (\lambda a.za)$$

Richtig

Ersetzungsregeln

Wenn ein Lambda-Ausdruck λx.<expr> auf einen Ausdruck E angewendet wird, werden alle freien Vorkommen von x in <expr> mit E ersetzt.

Wenn die Ersetzung eines freien Variablennamens von E in einen Ausdruck gebracht wird, indem dieser Name gebunden vorkommt, wird diese vor der Ersetzung umbenannt.

Beispiel: $(\lambda x.(\lambda y\ .(\mathbf{x}\ (\lambda x.xy)))) \ y$

$\Rightarrow (\lambda x.(\lambda t\ .(x\ (\lambda x.\ x\ t)))) \ y$

$\Rightarrow (\lambda t\ .(\mathbf{y}\ (\lambda x.\ x\ t)))$

Arithmetik

Zahlen müssen im Lambda-Kalkül wieder als Lambda-Abstraktionen definiert werden

Die Zahl Null kann wie folgt definiert werden:

Konvention:

$$\lambda s.(\lambda z.z) \equiv \lambda sz.z$$

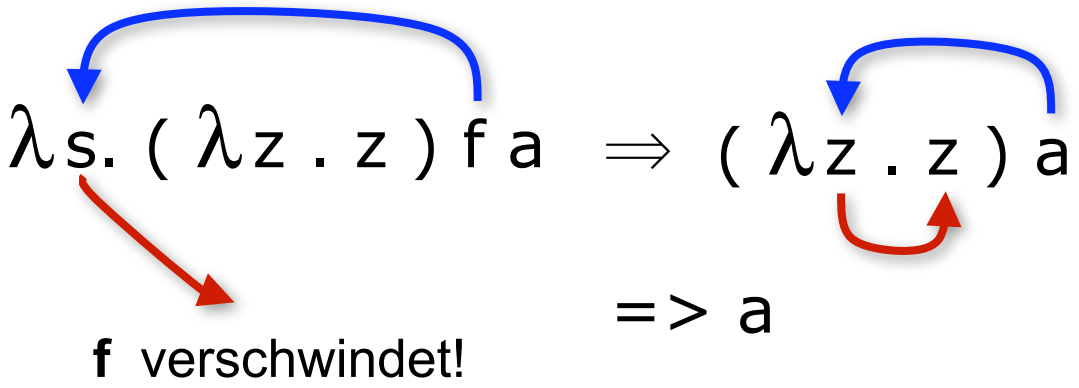
Wird als eine Funktion mit zwei Argumenten interpretiert

Wichtig ist, dass **s** innerhalb einer Applikation als erstes ersetzt wird.

Arithmetik

Wenn wir den Lambda-Ausdruck für Null auf eine Funktion mit einem Argument **a** verwenden, bekommen wir als Ergebnis nur **a**:

$$\lambda s.\,(\,\lambda z\,.\,z\,) \quad \equiv \quad 0$$



Arithmetik

Wir können dann die Zahlen mit folgenden Lambda-Ausdrücken darstellen:

$$\lambda s\ z\ .\ z \equiv 0$$
$$\lambda s\ z\ .\ s\ (z) \equiv 1$$
$$\lambda s\ z\ .\ s(s\ (z)) \equiv 2$$
$$\lambda s\ z\ .\ s(s(s(z))) \equiv 3$$
$$\ldots$$

f wird drei Mal auf das Argument **a** angewendet.

$$\lambda s\ z\ .\ s(s(s(z)))\ f\ a \Rightarrow \lambda z\ .\ f(f(f(z)))\ a$$
$$\Rightarrow f(f(f(a)))$$

Arithmetik

Wir definieren zuerst die Nachfolger-Funktion:

$\lambda w\, y\, x.\ y\ (w\ y\ x) \equiv S$

$S\ 0 \equiv (\lambda w\, y\, x.\ y\ (w\ y\ x))\ (\lambda s\ z\ .\ z)$

$\Rightarrow \lambda y\, x.\ y\ ((\lambda s\ z\ .\ z)\ y\ x)$

$\Rightarrow \lambda y\ x.\ y\ ((\lambda z\ .\ z)\ x)$

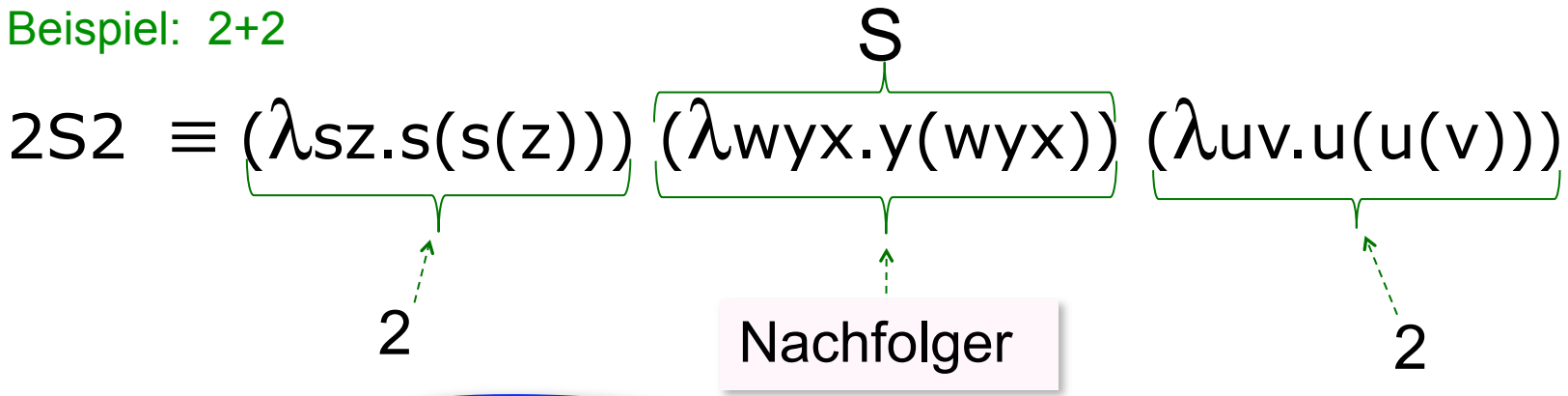
$\Rightarrow \lambda y\ x.\ y\ (x)$

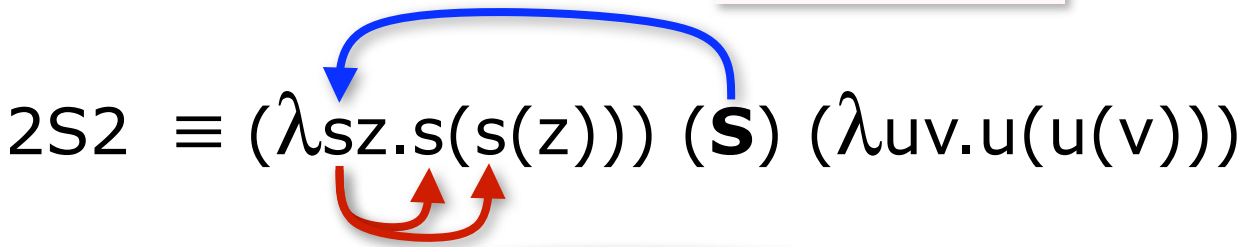
$\equiv 1$

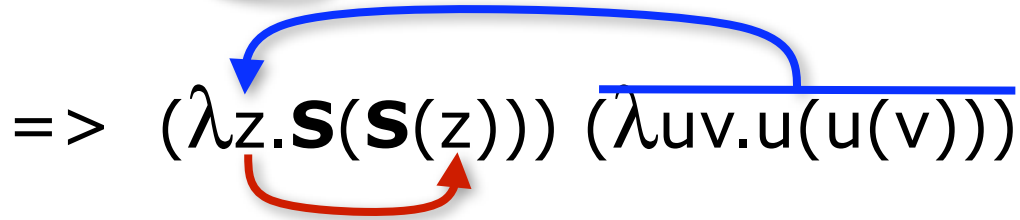
Arithmetik

Die Summe wird mit Hilfe der Nachfolger-Funktion definiert.

Beispiel: 2+2







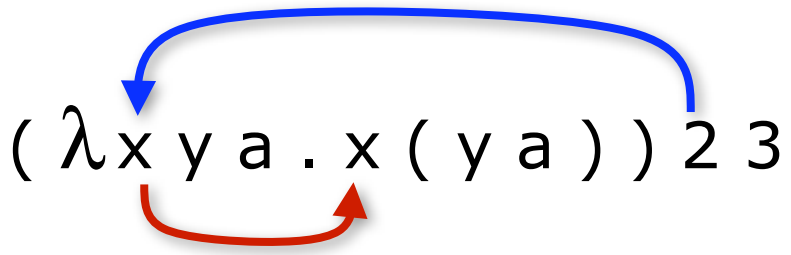
=> S(S(λuv.u(u(v)))) ≡ S(S2)

Multiplikation

Die Multiplikation kann mit folgendem Lambda-Kalkül-Ausdruck berechnet werden:

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

Beispiel: 2 * 3



$$=> (\ \lambda\ a\ .\ 2\ (\ 3\ a\ )\ )$$

Multiplikation

=> (λ a . 2 (3 a)) $\Rightarrow$ ... Tafel

Bedingungen

Definition der Wahrheitswerte:

$$\lambda x\ y\ .\ x \equiv T$$

$$\lambda x\ y\ .\ y \equiv F$$

Logische Operationen können wie folgt definiert werden:

AND-Funktion

$$\wedge \equiv \lambda x\ y\ .x\ y\ F$$

$$\vee \equiv \lambda x\ y\ .x\ T\ y$$

$$\neg \equiv \lambda x\ .\ x\ F\ T$$

Vergleich mit 0

Folgende Funktion ist wahr, wenn eine Zahl gleich Null ist, sonst ist sie falsch:

$$Z \equiv \lambda x \,.\, x\ F\ \neg F$$

Beispiel:

$$Z0 \equiv (\lambda x \,.\, x\ F\ \neg F)\ 0$$

$$\Rightarrow\ 0\ F\ \neg F$$

$$\Rightarrow\ \neg F$$

$$\Rightarrow\ T$$