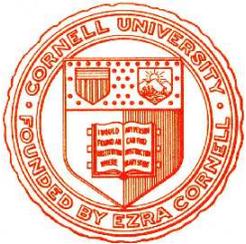


# Automatisierte Logik und Programmierung

## Teil II

### Konstruktive Typentheorie



# RÜCKBLICK – WO STEHEN WIR?

- **Wir sind fast da wo wir sein wollen**
  - Die Theorie abhängiger Typen ist sehr **ausdrucksstark**  
Umfaßt Prädikatenlogik,  $\lambda$ -Kalkül und Typzugehörigkeit
  - Die Theorie abhängiger Typen ist klein und **elegant**  
Nur 6 Regeln sind erforderlich (**symmetry**, **transitivity** sind simulierbar)  
Beweise von Meta-Eigenschaften bleiben überschaubar
- **Aber leider nur “fast”**
  - Das Konzept “Typsein” ist schwer zu formalisieren
  - Eine minimalistische Formalisierung ist inkonsistent  
“**Type**  $\in$  **Type**” ist nur möglich ohne abhängige Typen
  - Eine ausdrucksstarke Typentheorie muß(!) komplexer werden
- **Wir müssen eine neue Formulierung finden**
  - Konsistent und möglichst ausdrucksstark
  - Wir möchten bisherige Erkenntnisse weitestgehend wiederverwenden

**Wie gehen wir vor?**

# ES GIBT EINE STANDARDMETHODIK FÜR FORSCHUNG

- **Langfristige Ziele festlegen**
  - Was wollen wir insgesamt erreichen?
- **Bisherige Erkenntnisse aufsammeln**
  - Stand der Forschung, einschließlich eigener Ergebnisse
- **Alternative Lösungsansätze abwägen**
  - Welche grundsätzlichen Möglichkeiten gibt es, das Ziel zu erreichen?
- **Arbeitsplan entwickeln**
  - Eigenen Lösungsansatz skizzieren und an Beispielen analysieren
  - Kurz- und mittelfristige Arbeitspakete aufstellen
  - Wenn möglich, Meilensteine festlegen um Fortschritt zu messen

## Grundlage für Mathematik & Programmierung

- **Ausdrucksstarke formale Sprache**
  - Beschreibung mathematischer Konzepte und Aussagen
  - Beschreibung von Programmen und Berechnungsprozessen
  - Spezifikation von Software- und Systemeigenschaften
  - Formale Syntax möglichst flexibel und leicht verständlich
- **Akkurate, intuitiv einsichtige Semantik**
  - Präzisierung der Bedeutung formaler Ausdrücke  
möglichst nahe am “üblichen” Verständnis der formalen Notation
  - Konsistenz muß nachweisbar sein (um Vertrauen zu schaffen)
- **Inferenzkalkül für computergestützte Argumentation**
  - Formale Regeln für mathematische Beweisführung,  
sowie Verifikation, Synthese und Optimierung von Programmen
  - Nachweislich korrekt bezüglich der Semantik
  - Möglichst vollständig und leicht implementierbar

- **Wir haben Formalisierungen aller Aspekte**
  - Prädikatenlogik kann logische Struktur von Formeln analysieren
  - $\lambda$ -Kalkül ermöglicht Schlüsse über den Wert formaler Ausdrücke
  - Typentheorie unterstützt Schließen über Programmeigenschaften
  - Theorie abhängiger Typen vereinigt alle drei Aspekte
- **Problem ist konsistente Formalisierung von “Typsein”**
  - Beschreibung von  $T$  ist Typ durch  $T \in \mathbb{U}$  scheint sinnvoll
  - Aber abhängige Typen enthalten implizit Funktionen mit Zieltyp  $\mathbb{U}$   
Wenn diese Funktionen typisierbar sein sollen, muß  $\mathbb{U}$  ein Typ sein
  - Das Axiom  $\mathbb{U} \in \mathbb{U}$  macht die Theorie inkonsistent
- **Es gibt keine einfache Lösung**
  - Wir können weder auf abhängige Typen noch auf  $\mathbb{U}$  verzichten
  - Also müssen wir wieder zwischen Objekten und Typen unterscheiden
  - Dies kann entweder syntaktisch oder semantisch geschehen

- **Formalisierungen des Konzepts “Typsein”**
  - Formulierung abhängiger Typen benötigt Typen mit freien Variablen
  - Diese Ausdrücke müssen für jede Instanz der Variablen ein Typ sein
  - Die Präzisierung dieser Bedingung darf  $\mathbb{U} \in \mathbb{U}$  nicht erforderlich machen
- **Syntaktische Alternative** System F, Coq (Folie 6)
  - **Syntax-Trennung**: Variablen in Typausdrücken sind keine Objektvariablen  
Funktionen der Form  $\hat{T} \in S \rightarrow \mathbb{U}$  werden für  $x:S \rightarrow T[x]$  nicht benötigt
  - Alles andere kann im Prinzip beibehalten werden
- **Semantische Alternative** Martin-Löf (1978), Nuprl (Folie 7)
  - Keine syntaktische Trennung von Objekt- und Typausdrücken  
Der Unterschied liegt in der Bedeutung (rein **semantisches Konzept**)
  - Ausdrücke wie  $\mathbb{U}$  und  $S \rightarrow \mathbb{U}$  sind keine einfachen Typen wie  $\mathbb{N}$  oder  $S \rightarrow T$
  - Das **Universum gliedert sich syntaktisch in Stufen**  $\mathbb{U} \in \mathbb{U}_2 \in \mathbb{U}_3 \dots$   
Nur die Gesamtheit aller Stufen beschreibt das semantische “Typsein”

**Keine der Alternativen ist perfekt**

# (1) SYNTAKTISCHE ABTRENNUNG DES UNIVERSUMS

- **Spezielle Syntax vermeidet  $\mathbb{U}$  in Typausdrücken**
  - Strikte syntaktische Trennung von Objekt- und Typausdrücken
  - In  $x:S \rightarrow T[x]$  ist  $x$  eine sogenannte **Spezies Variable** und  $T[x]$  ein **Typ mit Parameter aus  $S$**
  - Diese Variablen benötigen andere Symbole als Objektvariablen
- **Girard's Paradox wird elegant umgangen**
  - Typisierte Objekte können selbst keine Typen enthalten
  - $\mathbb{U}$  ist selbst kein Typ, daher ist  $\mathbb{U} \in \mathbb{U}$  **nicht formulierbar**
  - Unklar ob andere Paradoxien formulierbar sind (unwahrscheinlich)
- **Theorie kann minimal bleiben**
  - Nur abhängiger Funktionenraum (“ $\Pi$ -Type”) erforderlich
  - Andere Konzepte wie bisher repräsentierbar
- **Beweisformalismus hat relativ komplexe Syntax**
  - Formulierung von Coq verbessert System F signifikant
  - Minimale Theorie zugunsten einer reichhaltigeren Syntax aufgegeben

## (2) SEMANTISCHE AUFTRENNUNG DES UNIVERSUMS

- **Semantisch der klarste Ansatz** Martin-Löf (1978)
  - Abstraktes “Typsein” ist von der Natur her ein semantisches Konzept
  - Einfache syntaktische Konzepte können das nicht vollständig erfassen
  - Ein Universum von Typen ist selbst ein Typ, aber kein einfacher Typ
  - Ein Universum, das  $\mathbb{U}$  enthält, ist ein “höheres” Universum (vgl. Menge von Mengen)
- **Führt zu Typhierarchie**  $\mathbb{U} \in \mathbb{U}_2 \in \mathbb{U}_3 \dots$ 
  - Typsein ist nur durch **Gesamtheit aller Universen** repräsentierbar
  - Komplikationen mit Mischtypen wie  $S \rightarrow \mathbb{U}$  lassen sich durch **kumulative Universen**  $\mathbb{U} \subseteq \mathbb{U}_2 \subseteq \mathbb{U}_3 \dots$  vermeiden
- **Formulierung der Theorie einfach, aber umfangreich**
  - Viele **Typkonstrukte** sind nicht in voller Allgemeinheit **simulierbar** und müssen explizit formalisiert werden
  - Inferenzregeln werden erst nach Präzisierung der Semantik formuliert
  - Aufwendig für Theorieentwickler, aber nützlich für Anwender
  - **Nuprl** fügt weitere Typkonstrukte zu Martin-Löf’s Theorie hinzu

# ENTWICKLUNG DER TYPENTHEORIE: PLAN

- **Minimale Kalküle sind elegant aber unnatürlich**
  - $\mathbb{B}, \mathbb{N}, \times, \wedge, \vee, \neg, \exists, =, \dots$  sind fundamentale mathematische Konzepte  
Eine Simulation erschwert formales Schließen in der Praxis
  - Martin-Löf's Ansatz ist klarer, auch wenn er (für uns) aufwendiger ist
- **Formuliere eine (konstruktive) semantische Theorie** (§8)
  - Methodik zur Präzisierung der Bedeutung formaler Ausdrücke
  - Syntaktische Darstellung des Kalküls wird Implementierungsaspekt  
(§12, ALuP II)
  - Formuliere Semantik bisheriger Konzepte ( $\rightarrow, \times, +, \{\}, \text{Logik}$ )
  - Untersuche grundlegende Meta-Eigenschaften
- **Formuliere Beweiskalkül auf Basis der Semantik** (§9)
  - Fixiere Struktur von Beweisen, Regeln und Evidenzkonstruktion  
Erfordert Formalisierung der Semantik von Universen und Gleichheit
  - Formuliere Regeln für  $\rightarrow, \times, +, \{\}, \mathbb{U}, =$  (und **Logik**)
  - Formalisierung weiter Konzepte kann hierauf aufsetzen (§10/11)

# Automatisierte Logik und Programmierung

## Einheit 8

### Martin-Löf's semantische Theorie



1. Semantikbeschreibung durch Urteile
2. Methodik zur Formulierung konkreter Typen
3. (Abhängige) Funktionen, Produkte & Summen
4. Logik in der Typentheorie
5. Grundsätzliche Eigenschaften der Theorie

# MATHEMATISCHE LOGIK IST MEHR ALS SYMBOLISMUS

- **In der Mathematik haben alle Ausdrücke eine Bedeutung**
  - Sie dienen der Beschreibung von Objekten, Funktionen, Mengen, ...
  - Uninterpretierte Symbole werden nur als Platzhalter verwendet
  - Bedeutung von Ausdrücken wie  $0$ ,  $3+4$ ,  $(2, 4)$ ,  $\lambda x.x$  ist intuitiv klar
  - Unspezifizierte Interpretation in Mengentheorie ist unintuitiv
- **Mathematische Logik sollte Bedeutung präzisieren**
  - Keine Trennung von syntaktischer Form und Semantik
  - Formale Ausdrücke bekommen eine  **feste Bedeutung**
  - Semantik macht implizites Verständnis der Symbole explizit
- **Typentheorie ist eine Grundagentheorie** (wie die Mengentheorie)
  - Semantik stützt sich “auf sich selbst” und intuitives Grundverständnis von (schematisierbaren) Aussagen wie “*wenn .. und .. dann*”
  - Nur die Widerspruchsfreiheit (Konsistenz) kann nachgewiesen werden

- **Unterscheide zwischen Aussagen und Urteilen**

- Logische Aussagen sind Sätze mit Bedeutung  
Formeln sind keine Aussagen, sondern syntaktische Repräsentationen
- Aussagen müssen nicht gültig sein (es sind nur Behauptungen)
- Die Festlegung “*Aussage A ist gültig*” (kurz “ $\vdash A$ ”) ist ein **Urteil**

- **Alle logischen Schlüsse beruhen auf Urteilen**

- Die Schlußfolgerung “*Aus A folgt  $A \vee B$* ” bedeutet in Wirklichkeit  
“*Wenn A gültig ist, dann ist auch  $A \vee B$  gültig*”
- $A \models A \vee B$  ist Kurzschreibweise für letzteres (nicht was das erste sagt)  
Man müsste eigentlich *A gültig  $\models A \vee B$  gültig* schreiben

- **Logische Schlüsse beinhalten implizite Typannahmen**

- Die Schlußfolgerung “*Aus A folgt  $A \vee B$* ” sagt implizit  
“*Wenn A und B Aussagen sind und A gültig ist, dann auch  $A \vee B$* ”
- Angemessen wäre also *A Aussage, B Aussage,  $\vdash A \models \vdash A \vee B$*

# TYPENTHEORIE BEHANDELT AUSDRÜCKE UND TYPEN

## ● Jedes Objekt hat einen Typ

- Schon bei der Beschreibung von Objekten wird der Typ verwendet  
Wir sagen “*Die Zahlen 1, 2, 3, ...*”, “*Die Funktion  $f$  mit  $f(x) = x^2$* ”,  
“*Die Menge  $M = \{x \in \mathbb{N} \mid x \geq 25\}$* ”, “*Die Aussage  $A$* ”
- Wir beschreiben Objekte durch mathematische Ausdrücke
- Ein Ausdruck ohne den zugehörigen Typ macht wenig Sinn

## ● Was macht einen Typ aus?

- Typen haben Objekte, die zu diesem Typ gehören  
Die Objekte 1, 2, 3, ... gehören zum Typ  $\mathbb{N}$
- Typen und Objekte benötigen Namen, die sie beschreiben  
1, 2, 3, .. und  $\mathbb{N}$  sind **kanonische** Ausdrücke (oft mit Objekt/Typ identifiziert)
- Objekte werden auch durch Operationen auf Objekten beschrieben  
Die **nichtkanonischen** Ausdrücke  $4+5$  und  $3*3$  beschreiben die Zahl 9

## ● Gleichheit ist essentiell

- Man muß sagen können, daß zwei Ausdrücke dasselbe beschreiben
- Gleichheit hängt immer vom Typ eines Objekts ab
- Man muß auch über Gleichheit von Typen reden können

# TYPENTHEORIE BENÖTIGT VIER ARTEN VON URTEILEN

- **Typ-Sein**

$T \text{ Typ}$

– Ausdruck  $T$  bezeichnet einen Typ (oder eine Menge)

- **Typgleichheit**

$S = T$

– Ausdrücke  $S$  und  $T$  bezeichnen den gleichen Typ

- **Typzugehörigkeit**

$t \in T$

– Ausdruck  $t$  beschreibt ein Element des durch  $T$  bezeichneten Typs

– Alternativ:  $t$  hat die durch  $T$  beschriebene Eigenschaft

- **Elementgleichheit**

$s = t \in T$

– Ausdrücke  $s$  und  $t$  beschreiben dasselbe Element des durch  $T$  bezeichneten Typs

**Alle anderen Urteilsarten sind hierdurch beschreibbar**

# LESARTEN DER TYPENTHEORETISCHEN URTEILE

Typen können sehr unterschiedlich interpretiert werden

$T$ Typ	$t \in T$	$\vdash T$
$T$ ist eine Menge	$t$ ist Element von $T$	$T$ ist nicht leer ( <i>inhabited</i> )
$T$ ist Aussage	$t$ ist Beweis für $T$	$T$ ist gültig (beweisbar)
$T$ ist Intention	$t$ ist Methode, $T$ zu erfüllen	$T$ ist erfüllbar (realisierbar)
$T$ ist Problem	$t$ ist Methode, $T$ zu lösen	$T$ ist lösbar
$T$ ist Spezifikation	$t$ ist Programm, das $T$ erfüllt	$T$ ist implementierbar

*Lesart der Typentheorie*

*Logische Sicht: "Propositionen als Datentypen"*

*Philosophische Sichtweise (Heyting)*

*Konstruktive Sicht (Kolmogorov)*

*Programmiersicht*

# WANN KÖNNEN KONKRETE URTEILE GEFÄLLT WERDEN?

- **Urteile sind Teil der Beschreibung eines Typs**

- Bezeichner (Ausdrücke) für Typen müssen deklariert werden
- Kanonische Elemente des Typs müssen beschrieben werden
- Es muß erklärt werden, wann zwei kanonische Elemente gleich sind

- **Einfach für elementare Ausdrücke**

- Typ-Sein wird für bestimmte Ausdrücke postuliert  $\mathbb{N} \text{ Typ}$
- Kanonische Elemente des Typs werden deklariert  $0 \in \mathbb{N}$
- Gleichheit kanonischer Elemente ist oft trivial  $0=0 \in \mathbb{N}$

- **Komplexer für zusammengesetzte Typen**

- Typ-Sein hat Voraussetzungen  $S \rightarrow T \text{ Typ, falls } S \text{ Typ und } T \text{ Typ}$
- Typzugehörigkeit ebenfalls  $\lambda x.t \in S \rightarrow T, \text{ falls } t \in T \text{ für alle } x \in S$
- Voraussetzungen für Gleichheit werden aufwendiger  
 $\lambda x_1.t_1 = \lambda x_2.t_2 \in S \rightarrow T, \text{ falls } t_1[s/x_1] = t_2[s/x_2] \in T \text{ für alle } s \in S$

- **Noch aufwendiger für abhängige Typen**

- Gleichheit von Typausdrücken ist nicht mehr unmittelbar  
 $x_1:S_1 \rightarrow T_1 = x_2:S_2 \rightarrow T_2, \text{ falls } T_1[s/x_1] = T_2[s/x_2] \text{ für alle } s \in S_1=S_2$

# WAS GESCHIEHT MIT NICHTKANONISCHEN AUSDRÜCKEN?

- **Nichtkanonische Ausdrücke sind keine Elemente**
  - Ein Ausdruck wie  $x+y$  oder  $f(x)$  hat keine Bedeutung
  - Bedeutung entsteht erst in Kombination mit kanonischen Elementen
  - Ausdrücke wie  $4+5$  oder  $(\lambda x.x)(4)$  haben einen **Wert**
  - Der Wert kann durch Reduktion bestimmt werden
  - **Urteile können nur über Werte von Ausdrücken gefällt werden**
- **Beschreibungen konkreter Urteile werden aufwendiger**
  - Ausdrücke müssen zunächst auf einen Wert reduziert werden
  - Die Bestimmung einer Normalform ist nicht erforderlich
  - Urteile können gefällt werden, wenn die **äußere Form kanonisch** ist
  - Ermöglicht Semantik für Ausdrücke, die nicht stark normalisierbar sind
  - **Der Wert von Ausdrücken wird durch Lazy Evaluation bestimmt**  
(Call-by-Name Strategie, die bei äußerer kanonischer Form terminiert)

## Semantische Beschreibung von Abhängigkeiten

- **Urteil wird unter Annahmen gefällt**
  - z.B. bei Gleichheit von  $\lambda$ -Termen  $t_1[s/x_1] = t_2[s/x_2] \in T$  für alle  $s \in S$   
Genauer: *Es gilt  $t_1[s/x_1] = t_2[s/x_2] \in T$  unter der Annahme  $s \in S$*   
Kurzschreibweise:  $s \in S \vdash t_1[s/x_1] = t_2[s/x_2] \in T$
  - Gültigkeit der Annahme wird vorausgesetzt (sonst wäre es kein Urteil)  
Nicht identisch mit “ $t_1[s/x_1] = t_2[s/x_2] \in T$  gilt, wenn  $s \in S$  gilt”
- **Annahmen sind nicht dasselbe wie Voraussetzungen**
  - Voraussetzungen sind Bedingungen, wann ein Urteil zu fällen ist
  - Bei abhängigen Konstrukten enthalten diese Bedingungen Annahmen
- **Allgemeine Form**  $x_1 \in T_1, \dots, x_n \in T_n [x_1 \dots x_{n-1}] \vdash J[x_1 \dots x_n]$ 
  - Urteil  $J$  und Typen  $T_i$  hängen von zuvor deklarierten Variablen ab
  - **Funktionalität**:  $J[t_i/x_i]$  gilt für alle kanonischen Elemente  $t_i \in T_i$
  - **Extensionalität**: für  $t_i = t'_i \in T_i$  gilt  $J[t_i/x_i]$  genau dann, wenn  $J[t'_i/x_i]$  gilt  
Auch wenn  $t_i$  oder  $t'_i$  nichtkanonisch sind
  - Gleichheitsannahmen werden durch Gleichheitstypen dargestellt (Einheit 9)

- **Definiere Ausdrücke (Terme) zur Beschreibung**
  - Unterteile Ausdrücke in kanonische und nichtkanonische Terme
  - Keine a-priori Trennung von Typ- und Elementausdrücken  
da in abhängigen Datentypen beide Formen gemischt vorkommen
- **Definiere den Wert von Ausdrücken**
  - Definiere Reduktion durch Angabe von Redizes und Kontrakta
  - Reduziere Ausdrücke mit Lazy Evaluation ( $\xrightarrow{l}$ ) auf kanonische Terme
- **Definiere, wann Urteile gefällt werden können**
  - Definiere Typ- und Elementgleichheit explizit für kanonische Terme
  - Definiere Gleichheit allgemeiner Terme mithilfe ihres Wertes  
Es gilt  $S=T$ , falls es kanonische Terme  $S'$  und  $T'$  gibt mit  
 $S \xrightarrow{l} S'$ ,  $T \xrightarrow{l} T'$  und  $S'=T'$  ist explizit definiert  
Es gilt  $s=t \in T$ , falls es kanonische Terme  $s'$ ,  $t'$  und  $T'$  gibt mit  
 $s \xrightarrow{l} s'$ ,  $t \xrightarrow{l} t'$ ,  $T=T'$  und  $s'=t' \in T'$  ist explizit definiert
  - Definiere Typzugehörigkeit / Typ-Sein als Abkürzung für Gleichheiten  
Es gilt  $t \in T$ , falls  $t=t \in T$  und  $T \text{ Typ}$ , falls  $T=T$

## Gleichheit und Wertbegriff sind “natürlich”

- **Gleichheiten sind  $S=T$  und  $s=t \in T$  sind transitiv und symmetrisch**
  - Reflexivität  $T = T$  gilt nur für (deklarierte) Typausdrücke  
 $t=t \in T$  gilt nur für (deklarierte) Ausdrücke vom Typ  $T$
  - Elementgleichheit ist transitiv und symmetrisch in  $s$  und  $t$   
Setzt voraus, daß konkrete Definitionen der Urteile nicht unsinnig gestaltet werden
- **Gleichheiten respektieren Reduktion**
  - $S=T$  gilt genau dann, wenn es ein  $S'$  gibt mit  $S \xrightarrow{l} S'$  und  $S'=T$
  - $s=t \in T$  gilt genau dann, wenn es ein  $s'$  gibt mit  $s \xrightarrow{l} s'$  und  $s'=t \in T$
- **Elementgleichheit impliziert Typ-Sein**
  - Wenn  $s=t \in T$  gilt, dann gilt auch  $T \text{ Typ}$
- **Gleiche Typen bzw. Elemente dürfen ausgetauscht werden**
  - Wenn  $S=T$  und  $s=t \in T$  gilt, dann gilt auch  $s=t \in S$
  - Gilt  $s_1=s_2 \in S$  und  $T[x] \text{ Typ}$  für alle  $x \in S$ , dann auch  $T[s_1/x]=T[s_2/x]$
  - Gilt  $s_1=s_2 \in S$  und  $t[x] \in T[x]$  für alle  $x \in S$ , dann auch  $t[s_1]=t[s_2] \in T[s_1]$

## Zentrales Konzept für Schließen über Berechnung

### ● Ausdrücke

Kanonisch:  $x : S \rightarrow T$       $x$  Variable,  $S$  und  $T$  Terme

$\lambda x . e$       $x$  Variable,  $e$  Term

Nichtkanonisch:  $e_1 e_2$       $e_1$  und  $e_2$  Terme

### ● Reduktion von Ausdrücken

$(\lambda x . u) t \xrightarrow{\beta} u[t/x]$

### ● Urteile für Typ- und Elementgleichheit

$x_1 : S_1 \rightarrow T_1 = x_2 : S_2 \rightarrow T_2$      falls  $S_1 = S_2$  und  $T_1[s_1/x_1] = T_2[s_2/x_2]$   
für alle Terme  $s_1, s_2$  mit  $s_1 = s_2 \in S_1$ .

$\lambda x_1 . t_1 = \lambda x_2 . t_2 \in x : S \rightarrow T$      falls  $x : S \rightarrow T$  Typ und  
 $t_1[s_1/x_1] = t_2[s_2/x_2] \in T[s_1/x]$   
für alle Terme  $s_1, s_2$  mit  $s_1 = s_2 \in S$

Annahme  $s_1 = s_2 \in S$  sichert Extensionalität des hypothetischen Urteils in Voraussetzung

## WAS IST ANDERS ALS ZUVOR?

- **Formulierung kommt ohne Universen aus**

- Typ-Sein von  $x:S \rightarrow T$  benötigt nur Typ-Sein von  $T[s/x]$  für  $x \in S$   
Urteil verlangt nicht, daß  $T$  eine Funktion von  $S$  in Typen ist
- Funktionalität und Extensionalität des hypothetischen Urteils sichert diesen Effekt für  $T[s/x]$  ohne explizite Verwendung von Funktionen
- **Funktionalität von  $T[s/x]$  ist semantische (!) Eigenschaft**

- **Girard's Paradox ist nicht formulierbar**

- Typ-Sein ist eine semantische Eigenschaft von Termen
- Wenn in einem syntaktischen Beweiskalkül Universen eingeführt werden (weil es sinnvolle Anwendungen in Formulierungen gibt), dann sind Universen selbst Typen aber keine Elemente von sich selbst
- Ein Term  $\mathbb{U}$  der formalen Sprache kann immer nur eine unvollständige Repräsentation des semantischen Konzepts Typ-Sein darstellen

## Gleichheit des syntaktischen Inhalts

- **Formale Rechtfertigung definitorischer Abkürzungen**
  - $\equiv$  ist Relation zwischen Termen (bzw. linguistischen Ausdrücken)  
z.B.  $A \Leftrightarrow B \equiv (A \Rightarrow B) \wedge (B \Rightarrow A)$
  - Gleichheit definiert zwei Ausdrücke als gegeneinander austauschbar
  - Üblicherweise gilt der linke Ausdruck als Abkürzung für den rechten
  - Unterstützt “konservative” Erweiterungen einer Sprache um neue Konzepte ohne Notwendigkeit für Erweiterung der Semantik
- **Nicht zu verwechseln mit Typ- oder Elementgleichheit**
  - Definitorische Gleichheit bezieht sich nur auf die äußere Form  
Es ist leicht zu testen, ob zwei Ausdrücke definitorisch gleich sind
  - Typ- und Elementgleichheit beziehen sich auf Bedeutung bzw. Wert  
Semantische Gleichheit ist im Allgemeinen unentscheidbar

# ANWENDUNGEN DES ABHÄNGIGEN FUNKTIONENRAUMS

- **Funktionsraum**  $S \rightarrow T \equiv x:S \rightarrow T$ 
  - Funktionsraum ohne Abhängigkeiten ist einfacher Spezialfall
  - $T$  darf nicht von  $x$  abhängen
  - Alle weiteren Aspekte sind identisch
- **(Getypter) Allquantor**  $\forall x:T.P[x] \equiv x:T \rightarrow P[x]$ 
  - Basiert auf logischer Sicht der “Propositionen als Datentypen”  
Eine logische Aussage entspricht dem Typ ihrer Evidenzen
  - Notation entspricht informatiktypischer Schreibweise
  - Quantifiziert wird über alle Objekte des Typs  $T$
- **Implikation**  $A \Rightarrow B \equiv A \rightarrow B$ 
  - Definition stützt sich auf Einsichten der Curry-Howard Isomorphie
- **$\Pi$ -Type**  $\Pi x \in S.T[x] \equiv x:S \rightarrow T[x]$ 
  - Kartesisches Produkt einer Familie von Typen
  - Basiert auf Sicht von Funktionen als Menge von Paaren (Punkten)
- **Record-Strukturen in Programmiersprachen**
  - Records bilden Labels in Typen ab, die vom konkreten Label abhängen

# RECORD TYPES ALS ABHÄNGIGER FUNKTIONENRAUM

- **Record-Strukturen verbinden verschiedenartige Daten**

- z.B. Deklaration eines Typs und Definition eines Objekts in OCaml

```
type account
= { first : string
  ; last : string
  ; age : int
  ; balance : float
  }
```

```
let pers
= { first = "John"
  ; last = "Doe"
  ; age = 150
  ; balance = 0.12
  }
```

- Typ einer Komponente hängt ab vom Wert des Labels

- **Simulation durch abhängigen Funktionenraum**

- Definiere `account` mit einer typ-wertigen Funktion

```
acc-component-of
≡ λl. if l='first' then String
     else if l='last' then String
     else if l='age' then ℤ
     else if l='last' then Float
     else {}
```

```
acc-component-of ∈ Label → U
```

```
account
≡ l:Label
  → acc-component-of l
account ∈ U
```

- Die Definition von `pers` ist analog zu der von `acc-component-of`

Der Einfachheit halber nehmen wir an, daß `String`, `ℤ`, `Float`, `U`, `{}` bereits definiert sind

**Zeige**  $\lambda f . \lambda x . f x \in (S \rightarrow T) \rightarrow S \rightarrow T$  für beliebige  $S, T$

## Analysiere Bedingungen der semantischen Urteile

- $\lambda f . \lambda x . f x$  ist kanonisch und es gilt  $(t = t \in T$  wird durch  $t \in T$  abgekürzt)  
 $\lambda f . \lambda x . f x \in (S \rightarrow T) \rightarrow S \rightarrow T$  falls  $\lambda x . g x \in S \rightarrow T$  für alle  $g \in S \rightarrow T$
- $\lambda x . g x$  ist kanonisch und es gilt  $\lambda x . g x \in S \rightarrow T$   
falls  $g s \in T$  für alle Terme  $g \in S \rightarrow T, s \in S$
- $g s$  ist nicht kanonisch und kann nicht reduziert werden  
aber es wurde die Annahme  $g \in S \rightarrow T$  gemacht
- Das Urteil  $g \in S \rightarrow T$  bedeutet, daß es ein kanonisches Element  $g'$   
gibt mit  $g \xrightarrow{l} g'$ , und  $g' \in S \rightarrow T$  ist explizit definiert
- $g' \in S \rightarrow T$  ist nur explizit definiert für Terme der Form  $g' = \lambda y . t$   
 $\lambda y . t \in S \rightarrow T$  gilt nur, wenn  $t[s/y] \in T$  für alle Terme  $s \in S$  gilt
- Aus der Annahme  $g \in S \rightarrow T$  folgt also die Existenz eines Terms  $t$  mit  
 $g s \xrightarrow{l} (\lambda y . t) s \xrightarrow{l} t[s/y] \in T$  für alle Terme  $s \in S$   
und somit ist  $g s \in T$  für alle Terme  $s \in S$



**Zeige  $\lambda x. x x \notin S \rightarrow T$  für jeden (kanonischen) Typ  $S, T$**

## Analysiere Bedingungen der semantischen Urteile

- $\lambda x. x x$  ist kanonisch und es gilt  
 $\lambda x. x x \in S \rightarrow T$  falls  $s s \in T$  für alle Terme  $s \in S$
- $s s$  ist nicht kanonisch und kann zu einem Wert reduziert werden, wenn  $s$  zu einem Ausdruck der Form  $\lambda y. t$  reduziert werden kann
- Dies ist nur möglich, wenn  $S = Y \rightarrow Z$  für gewisse Typen  $Y, Z$  ist und  $t[s/y] \in Z$  für alle  $s \in Y$  gilt  
In dem Fall folgt  $s s \xrightarrow{l} (\lambda y. t) s \xrightarrow{l} t[s/y] \in Z$  falls  $s \in Y$   
Damit muß  $Z = T$  und  $S = Y$ , also  $S = S \rightarrow T$  sein
- Für die Behauptung  $S = S \rightarrow T$  gibt es keine Unterstützung  
Damit gibt es auch keine Rechtfertigung, das Urteil  $s s \in T$  zu fällen  
Die Sprechweise “*Das Urteil ist nicht gültig*” sollte vermieden werden
- **Das Urteil  $\lambda x. x x \in S \rightarrow T$  kann nicht gefällt werden** ✓

# (NEU-)FORMULIERUNG ABHÄNGIGER PRODUKTRÄUME

## Fundamentale Strukturierung von Daten

- **Ausdrücke**

Kanonisch:  $x : S \times T$   $x$  Variable,  $S$  und  $T$  Terme

$(e_1, e_2)$   $e_1$  und  $e_2$  Terme

Nichtkanonisch:  $\text{match } e \text{ with } \langle x, y \rangle \mapsto u$   $x, y$  Variablen,  $e$  und  $u$  Terme

- **Reduktion von Ausdrücken**

$\text{match } (s, t) \text{ with } \langle x, y \rangle \mapsto u \xrightarrow{\beta} u[s, t/x, y]$

- **Urteile für Typ- und Elementgleichheit**

$x_1 : S_1 \times T_1 = x_2 : S_2 \times T_2$  falls  $S_1 = S_2$  und  $T_1[s_1/x_1] = T_2[s_2/x_2]$   
für alle Terme  $s_1, s_2$  mit  $s_1 = s_2 \in S_1$ .

$(s_1, t_1) = (s_2, t_2) \in x : S \times T$  falls  $x : S \times T$  Typ und  
 $s_1 = s_2 \in S$  und  $t_1 = t_2 \in T[s_1/x]$

Annahme  $s_1 = s_2 \in S$  sichert Extensionalität des hypothetischen Urteils in Voraussetzung

# ANWENDUNGEN DES ABHÄNGIGEN PRODUKTRAUMS

- **Produktraum**  $S \times T \equiv x:S \times T$ 
  - Einfacher Spezialfall –  $T$  darf nicht von  $x$  abhängen
  - Weitere Abkürzungen  $e.1 \equiv \text{match } e \text{ with } \langle x, y \rangle \mapsto x$   
 $e.2 \equiv \text{match } e \text{ with } \langle x, y \rangle \mapsto y$
- **(Getypter) Existenzquantor**  $\exists x:T.P[x] \equiv x:T \times P[x]$ 
  - Basiert auf logischer Sicht der “Propositionen als Datentypen”
  - Notation entspricht informatiktypischer Schreibweise
  - Quantifiziert wird über alle Objekte des Typs  $T$
- **Konjunktion**  $A \wedge B \equiv A \times B$ 
  - Definition stützt sich auf Einsichten der Curry-Howard Isomorphie
- **$\Sigma$ -Type**  $\Sigma_{x \in S}.T[x] \equiv x:S \times T[x]$ 
  - Disjunkte Vereinigung einer Familie von Typen
  - Basiert auf Sicht von Injektionen als Objekt-Label Paare
- **Modulstrukturen in Programmiersprachen**
  - Typen der Operationen hängen ab von deklarierten Namen

**Zeige**  $\lambda p. (p.1) \in (S \times T) \rightarrow S$  für alle  $S, T$

## Analysiere Bedingungen der semantischen Urteile

- $\lambda p. (p.1)$  ist kanonisch und es gilt  
 $\lambda p. (p.1) \in (S \times T) \rightarrow S$  falls  $e.1 \in S$  für alle Terme  $e \in S \times T$
- $e.1$  ist nicht kanonisch und kann nicht reduziert werden  
aber es wurde die Annahme  $e \in S \times T$  gemacht
- Das Urteil  $e \in S \times T$  bedeutet, daß es ein kanonisches Element  $e'$  gibt mit  $e \xrightarrow{l} e'$ , und  $e' \in S \times T$  ist explizit definiert
- $e' \in S \times T$  ist nur explizit definiert für Terme der Form  $e' = (s, t)$   
 $(s, t) \in S \times T$  gilt nur, wenn  $s \in S$  und  $t \in T$  gilt
- Aus der Annahme  $e \in S \times T$  folgt also die Existenz zweier Terme  $s, t$  mit  
$$e.1 \xrightarrow{l} (s, t).1 \xrightarrow{l} s \in S$$
  
und somit ist  $e.1 \in S$  für alle Terme  $e \in S \times T$



# (NEU-)FORMULIERUNG DES SUMMENTYPS

## Disjunkte Vereinigung der Elemente zweier Typen

### • Ausdrücke

Kanonisch:  $S+T$   $S$  und  $T$  Terme  
 $\text{inl}(e)$   $e$  Term  
 $\text{inr}(e)$   $e$  Term

Nichtkanonisch:  $\text{case } e \text{ of } \text{inl}(x_1) \mapsto e_1 \mid \text{inr}(x_2) \mapsto e_2$   
 $x_1, x_2$  Variablen,  $e, e_1$  und  $e_2$  Terme

### • Reduktion von Ausdrücken

$\text{case } \text{inl}(s) \text{ of } \text{inl}(x_1) \mapsto e_1 \mid \text{inr}(x_2) \mapsto e_2 \xrightarrow{\beta} e_1[s/x_1]$   
 $\text{case } \text{inr}(t) \text{ of } \text{inl}(x_1) \mapsto e_1 \mid \text{inr}(x_2) \mapsto e_2 \xrightarrow{\beta} e_2[t/x_2]$

### • Urteile für Typ- und Elementgleichheit

$S_1 + T_1 = S_2 + T_2$  falls  $S_1 = S_2$  und  $T_1 = T_2$   
 $\text{inl}(s_1) = \text{inl}(s_2) \in S+T$  falls  $S+T$  Typ und  $s_1 = s_2 \in S$   
 $\text{inr}(t_1) = \text{inr}(t_2) \in S+T$  falls  $S+T$  Typ und  $t_1 = t_2 \in T$

## Fallunterscheidungen und Aufzählung von Alternativen

- **Disjunktion**  $A \vee B \equiv A + B$ 
  - Definition stützt sich auf Einsichten der Curry-Howard Isomorphie
- **Boolescher Typ als Spezialfall**
  - Vereinigung zweier einelementiger Typen:  $\mathbb{B} \equiv \text{Unit} + \text{Unit}$
  - Durch Marker entstehen genau zwei unterscheidbare Elemente  
 $\mathbf{T} \equiv \text{inl}()$ ,  $\mathbf{F} \equiv \text{inr}()$
  - Nichtkanonisches Element analysiert nur den Marker  
 $\text{if } b \text{ then } s \text{ else } t \equiv \text{case } b \text{ of inl}(x) \mapsto s \mid \text{inr}(y) \mapsto t$
- **Aufzählungstypen als syntaktische Verallgemeinerung**
  - Wichtiges Konzept vieler moderner Programmiersprachen
  - Es werden viele Marker unterschieden (nicht nur zwei)
  - Nichtkanonisches Element entspricht komplexer Fallunterscheidung

# SEMANTISCHE UNTERSUCHUNG VON EIGENSCHAFTEN IV

**Für alle  $S, T$  gilt**  $\lambda z. \text{ case } z \text{ of } \text{inl}(x) \mapsto x \mid \text{inr}(y) \mapsto (y, y)$   
 $\in z : S + T \rightarrow \text{case } z \text{ of } \text{inl}(x) \mapsto S \mid \text{inr}(y) \mapsto T \times T$

## Analysiere Bedingungen der semantischen Urteile

- $\lambda z. \text{ case } z \text{ of } \text{inl}(x) \mapsto x \mid \text{inr}(y) \mapsto (y, y)$  ist kanonisch und gehört zu  $z : S + T \rightarrow \text{case } z \text{ of } \text{inl}(x) \mapsto S \mid \text{inr}(y) \mapsto T \times T$   
falls  $\text{case } e \text{ of } \text{inl}(x) \mapsto x \mid \text{inr}(y) \mapsto (y, y)$   
 $\in \text{case } e \text{ of } \text{inl}(x) \mapsto S \mid \text{inr}(y) \mapsto T \times T$  für alle  $e \in S + T$
- $\text{case } e \text{ of } \text{inl}(x) \mapsto x \mid \text{inr}(y) \mapsto (y, y)$  ist nicht kanonisch und kann nicht reduziert werden aber es wurde die Annahme  $e \in S + T$  gemacht
- Wegen  $e \in S + T$  gibt es ein kanonisches  $e'$  mit  $e \xrightarrow{l} e'$ , und  $e' \in S + T$  ist explizit definiert
- $e' \in S + T$  ist nur explizit definiert für  $e' = \text{inl}(s)$  mit  $s \in S$  und  $e' = \text{inr}(t)$  mit  $t \in T$
- Aus der Annahme  $e \in S + T$  folgt also die Existenz eines Terms  $s$  mit

$\text{case } e \text{ of } \text{inl}(x) \mapsto x \mid \text{inr}(y) \mapsto (y, y)$   
 $\xrightarrow{l} \text{case } \text{inl}(s) \text{ of } \text{inl}(x) \mapsto x \mid \text{inr}(y) \mapsto (y, y) \xrightarrow{l} s \in S$

und  $\text{case } e \text{ of } \text{inl}(x) \mapsto S \mid \text{inr}(y) \mapsto T \times T$   
 $\xrightarrow{l} \text{case } \text{inl}(s) \text{ of } \text{inl}(x) \mapsto S \mid \text{inr}(y) \mapsto T \times T \xrightarrow{l} S$

oder eines Terms  $t$  mit analogen Eigenschaften und somit ist

$\text{case } e \text{ of } \text{inl}(x) \mapsto x \mid \text{inr}(y) \mapsto (y, y) \in \text{case } e \text{ of } \text{inl}(x) \mapsto S \mid \text{inr}(y) \mapsto T \times T$  ✓

- **Logik kann in Typstruktur eingebettet werden**
  - Prinzip “*Propositionen als Datentypen*” macht Aussagen zu Typen
  - Trennung zwischen Aussage  $A$  und Typ  $[A]$  der Evidenzen entfällt
  - Evidenz für Aussage  $A$  ist Element des Typs  $A$
- **Konnektive entsprechen Typkonstruktoren** (vgl. §3, Folie 24)
  - Allquantor entspricht abhängigem Funktionenraum
  - Implikation entspricht unabhängigem Funktionenraum
  - Existenzquantor entspricht abhängigem Produktraum
  - Konjunktion entspricht unabhängigem Produktraum
  - Disjunktion entspricht Summentyp
  - Negation entspricht Funktionenraum mit leerem Bildbereich
- **Logik wird konservative Erweiterung der Typentheorie**
  - Konnektive werden durch definitorische Gleichheiten eingebettet
  - Eine echte Erweiterung der Theorie ist nicht erforderlich

## • Definitiorische Gleichheiten für Konnektive

$$P \wedge Q \equiv P \times Q$$

$$P \vee Q \equiv P + Q$$

$$P \Rightarrow Q \equiv P \rightarrow Q$$

$$\neg P \equiv P \rightarrow f$$

$$f \equiv \{\}$$

$$\exists x:T. P[x] \equiv x:T \times P[x]$$

$$\forall x:T. P[x] \equiv x:T \rightarrow P[x] \quad (\text{Alternative: } \bigcap x:T. P[x] \text{ siehe §11})$$

Leerer Datentyp zur Beschreibung von  $f$  muß noch formalisiert werden

## • Definitionen liefern getypte Logik

(erweitert Logik aus §2/3)

– Quantoren beziehen sich auf Elemente konkreter Typen (statt vages “ $\cup$ ”)

– Syntax “ $\forall x:T. P$ ” ist informatiktypisch (statt “ $(\forall x)P$ ”)

– **Prioritäten** zwischen Konnektiven sparen Klammern

$\neg$  bindet stärker als  $\wedge$ , dann folgt  $\vee$ , dann  $\Rightarrow$ , dann  $\exists$ , dann  $\forall$

$\Rightarrow$  ist rechtsassoziativ

# FORMALISIERUNG EINES LEEREN DATENTYPS $\{\}$

- **Datentyp ohne Elemente**

- Leerheit bedeutet, daß  $\{\}$  keine kanonischen Elemente haben darf
- $\{\}$  ist Gegenstück zum logischen Konzept der Falschheit

- **Eigenschaften von Kombinationen mit  $\{\}$**

- $\{\} \times T$  ist ebenfalls ein Datentyp ohne Elemente, denn  $(s, t) \in \{\} \times T$  impliziert  $s \in \{\}$  und  $t \in T$
- $\{\} + T$  ist isomorph zum Typ  $T$  (aber nicht gleich!) denn  $\text{inl}(s) \in \{\} + T$  impliziert  $s \in \{\}$  und  $\text{inr}(t) \in \{\} + T$  genau dann, wenn  $t \in T$
- $T \rightarrow \{\}$  muß leer sein, wenn  $T$  nicht leer ist (entspricht Negation) denn  $\lambda x. a \in T \rightarrow \{\}$  impliziert  $a[t/x] \in \{\}$  für alle  $t \in T$
- $\{\} \rightarrow T$  muß Elemente der Form  $\lambda x. t$  für beliebige  $t$  haben, denn  $\lambda x. t \in \{\} \rightarrow T$  falls  $\{\} \rightarrow T$  Typ und  $t[s/x] \in T$  für alle  $s \in \{\}$

- **$\{\}$  braucht ein nichtkanonisches Element**

- $\text{any}(x)$  mit  $\text{any}(x) \in T$  für  $x \in \{\}$

- **Einbettung durch Definitorische Gleichheit ist möglich**

Es gibt viele Aussagen, die prinzipiell ohne Evidenz sein müssen

- $\forall P:U. P$  “alles ist wahr” darf nicht gelten
- $\forall X:U. X=X \rightarrow X$  in  $U$  Ein Typ kann nicht sein eigener Funktionenraum sein
- $0=1$  in  $\mathbb{N}$  in diesem Fall wären alle Zahlen gleich
- $\mathbf{T}=\mathbf{F}$  in  $\mathbb{B}$  in diesem Fall wären alle Terme gleich

Datentypen für Universen, Gleichheit, Zahlen und  $\mathbb{B}$  werden in §§9–11 eingeführt

Ausdrücke könnten für konservative Erweiterung verwendet werden

- **Definition von  $\{\}$  als Abkürzung ist problematisch**

- Konservative Erweiterung liefert kein festes nichtkanonisches Element
- Ein Term  $any(x)$  mit  $any(x) \in T$  für  $x \in \{\}$  ist nicht definierbar

- **Definition als primitiver Datentyp zu bevorzugen**

- Leer-Sein ist ein grundlegendes mathematisches Konzept

## Datentyp ohne Elemente

- **Ausdrücke**

Kanonisch:  $\{\}$

Nichtkanonisch:  $\text{any}(e)$   $e$  Term

- **Reduktion von Ausdrücken**

– entfällt, da keine kanonischen Elemente von  $\{\}$  definiert

- **Urteile für Typ- und Elementgleichheit**

$$\{\} = \{\}$$

$$e_1 = e_2 \in \{\} \quad \textit{gilt niemals}$$

Das Urteil  $e=e \in \{\}$  darf niemals gefällt werden

# LEERE DATENTYPEN ERMÖGLICHEN SELTSAME TYPEN

- **$x:\{\} \rightarrow T$  ist ein Typ, auch wenn  $T$  kein Typ ist**
  - Man kann auch  $T \equiv \lambda x. \lambda y. y$  oder beliebige andere Ausdrücke wählen
  - Es gilt  $x:\{\} \rightarrow T$  Typ, falls  $\{\}$  Typ und  $T[s]$  Typ für alle  $s \in \{\}$   
Da  $s \in \{\}$  niemals gilt, ist die Bedingung immer erfüllt also  $x:\{\} \rightarrow T$  Typ
- **Der Term  $t = \lambda z. (\lambda x. x x) (\lambda x. x x)$  wird typisierbar**
  - Es gilt  $t \in \{\} \rightarrow T$  für jedes beliebige  $T$ , denn  
 $t \in \{\} \rightarrow T$  falls  $\{\} \rightarrow T$  Typ und  $(\lambda x. x x) (\lambda x. x x) \in T$  für alle  $s \in \{\}$   
Da  $s \in \{\}$  niemals gilt, ist die Bedingung immer erfüllt
- **Nicht alle typisierbaren Terme sind normalisierbar**
  - Ein leerer Datentyp führt zum Verlust der starken Normalisierbarkeit
  - Es gilt nicht einmal schwache Normalisierbarkeit

Terme, die durch Beweisführung generiert werden ( $\vdash \S 10$ ), sind stark normalisierbar  
Terme mit Selbstreferenzen, wie  $\lambda z. (\lambda x. x x) (\lambda x. x x)$ , können nicht generiert werden

# TYPENTHEORIE BENÖTIGT LAZY EVALUATION

- **Starke Normalisierbarkeit ist unhaltbare Anforderung**
  - Simulationen von  $\{\}$  erlauben ebenfalls nichtterminierende Terme
  - Propositionale Gleichheit, Universen ( $\mapsto$  §10) und  $\{\}$  sind essentiell
  - SN wäre nur bei Verzicht auf alle drei Konzepte erreichbar
- **“Klassische” Normalisierung macht nicht wirklich Sinn**
  - $\lambda z.(\lambda x. x x)(\lambda x. x x)$  hat keine Normalform im bisherigen Sinn
  - $\lambda z.(\lambda x. x x)(\lambda x. x x)$  stellt dennoch eine “echte Funktion” dar
    - Für jedes  $s \in \{\}$  bestimmt sie einen Wert aus  $T$
    - Da es kein  $s \in \{\}$  gibt, muß  $(\lambda x. x x)(\lambda x. x x)$  nie berechnet werden
  - $\lambda z.(\lambda x. x x)(\lambda x. x x)$  **terminiert, wenn man nur ‘bei Bedarf’ reduziert**
- **Lazy Evaluation paßt besser zur Semantik**
  - Reduktion kann terminieren, sobald die äußere Form kanonisch ist
  - Die Semantik kanonischer Terme ist explizit definiert
  - Es ist nicht erforderlich, auch die Teilterme zu normalisieren, um die Bedeutung eines kanonischen Terms zu erklären

# KONSEQUENZEN DER EINBETTUNG DER LOGIK

- **Typentheorie umfaßt intuitionistische Logik**
  - Konservative Erweiterung (Folie 25) ergibt Logik aus §§2/3
  - Einziger Unterschied ist zusätzliche Angabe des Typs von Objekten
- **Klassische Logik ist simulierbar**
  - Allquantor, Konjunktion, und Negation definiert wie zuvor
  - Definition von  $\vee$ ,  $\Rightarrow$ ,  $\exists$  ohne “konstruktiven Inhalt”

$$\frac{P \vee_c Q}{P \vee_c Q} \equiv \neg(\neg P \wedge \neg Q)$$

$$\frac{P \Rightarrow_c Q}{P \Rightarrow_c Q} \equiv \neg P \vee_c Q$$

$$\frac{\exists_c x:T. P}{\exists_c x:T. P} \equiv \neg(\forall x:T. \neg P)$$

Ersetzt man zusätzlich jede atomare Teilformel  $P$  durch  $\neg\neg P$ , dann wird jede klassisch gültige Formel auch gültig in der Typentheorie

- **Weitere Logiken sind ebenfalls konservativ einbettbar**
  - Ungetypte Logik (verwende einen Top-Typ als Default)
  - Prädikatenlogik beliebiger Stufe (gleiche Definition wie zuvor)
  - Funktionenkalküle höherer Ordnung

## Entwurfsentscheidungen wurden (oft implizit) getroffen

- **Die Semantik der Theorie steht im Vordergrund**
  - Bedeutung von Konzepten wird durch semantische Urteile definiert
  - Syntaktische Darstellung im Beweiskalkül muß Semantik reflektieren
  - In anderen Kalkülen steht Semantik (wenn überhaupt) im Hintergrund
- **Die Semantik der Theorie basiert auf Lazy Evaluation**
  - + Urteile können gefällt werden, wenn die äußere Form kanonisch ist
  - + Terminierende Anwendungen des Y-Kombinators werden typisierbar
- **Keine syntaktische Trennung von Objekten und Typen**
  - + Typ-Sein und -Zugehörigkeit ist eine semantische Eigenschaft
  - + Keine Einschränkung für Verwendung von Symbolen durch Anwender
  - Erfordert explizite Behandlung des Typseins im Beweiskalkül
- **Gleichheit ist semantisch und damit extensional**
  - + Natürliche, wert-basierte Gleichheit in Theorie verankert
  - Extensionale Sicht bringt viele Unentscheidbarkeiten in die Theorie

# GRUNDSÄTZLICHE EIGENSCHAFTEN DER THEORIE II

- **Typentheorie ist eine (konstruktive) Grundlagentheorie**
  - + Semantik formuliert natürliches Verständnis von Konzepten  
Keine Interpretation in anderen Theorien erforderlich
- **Typentheorie ist offen für Erweiterungen**
  - + Neue Typkonstrukte können explizit hinzugefügt werden  
(wenn sie im Stil der Theorie unabhängig formulierbar sind)
  - + Keine Einschränkung auf konservativ simulierbare Konzepte
  - Selbstreflektion ist begrenzt: Schließen über alle Terme unmöglich  
Derartige Beweise würden bei Erweiterungen ungültig
- **Umfangreicher Kalkül**
  - + **Gut für Anwender**: wichtige Grundkonzepte sind explizit unterstützt
  - Viele Namen für Terme und Regeln müssen erlernt werden
  - **Aufwendig für Entwickler**: viele Fälle in Beweisen von Eigenschaften

**Es gibt gute Gründe für diese Entscheidungen**

aber wenn man andere Ziele hat, würde man andere Entscheidungen fällen

# WELCHE ALTERNATIVEN HÄTTEN WIR AUCH GEHABT?

- **Syntaktische Behandlung des Typseins**

- + : Einfache Repräsentation des Typuniversums

- : Objekte und Typen müssen syntaktisch getrennt werden

- **Intensionale Gleichheit**

- + : Leicht zu testen, da rein syntaktisches Konzept (textliche Identität)

- : Semantische Gleichheit muß vom Benutzer definiert werden

- **Minimale Kalküle sind mathematisch elegant**

- + : Gut für Entwickler: Nachweis von Eigenschaften leichter

- + : Leicht zu erlernen: Regelsatz ist überschaubar

- : Mathematische Grundkonzepte müssen (korrekt!) simuliert werden

- Großer Formalisierungsaufwand für (Erst-)Anwender

- : Einfache Schlüsse verlangen viele elementare Inferenzen (Effizienz?)

- **Geschlossene Formalisierung**

- (Theorie wird ein für alle Mal festgelegt)

- + : Erlaubt Beweise mit Fallunterscheidung über alle möglichen Terme

- : Alle nicht explizit definierten Konzepte müssen simulierbar sein

- Teile von Mathematik & Programmierung nicht ausdrückbar ?

## Erweiterungen werden immer wieder nötig sein

- **Beschreibe Syntax**
  - Ausdrücke die zur Formulierung des Typs verwendet werden sollen
  - Ausdrücke zur Erzeugung und Verwendung von Objekten des Typs
  - Keine Trennung dieser Ausdrücke auf syntaktischer Ebene
- **Trenne kanonische und nichtkanonische Terme**
  - Beschreibe Auswertung nichtkanonischer Terme durch explizite Angabe von Redizes und der zugehörigen Kontrakta
- **Beschreibe Semantik kanonischer Terme durch Urteile**
  - Definiere Voraussetzungen, um Urteile über Typ-Gleichheit und Gleichheit kanonischer Elemente fällen zu können
  - Vermeide Verwendung von Termen die zu anderen Typkonzepten gehören, um deren Semantik nicht unabsichtlich zu beeinflussen
  - Prüfe Konsistenz der Ergänzung mit bisheriger Theorie