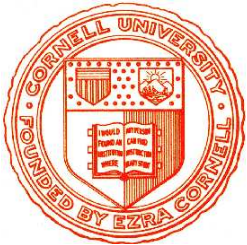
Automatisierte Logik und Programmierung

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, λ-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 ausdruckstarke Typentheorie muß(!) komplexer werden
- **Wir müssen eine neue Formulierung finden**
  - Konsistent und möglichst ausdruckstark
  - 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

Typentheorie: Ziele

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

Typentheorie: Erkenntnisse

- **Wir haben Formalisierungen aller Aspekte**
  – Prädikatenlogik kann logische Struktur von Formeln analysieren
  – λ-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

Typentheorie: Alternative Lösungsansätze

- **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 $\hat{T} \in S \to \mathbb{U}$ werden für $x{:}S \to T[x]$ 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 $S \to \mathbb{U}$ sind keine einfachen Typen wie $\mathbb{N}$ oder $S \to T$
  - Das Universum gliedert sich syntaktisch in Stufen $\mathbb{U} \in \mathbb{U}_2 \in \mathbb{U}_3 \ldots$
    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 ("Π-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

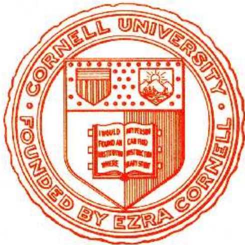
- **Semantisch der klarste Ansatz** Martin-Löf (1978)
  - Abstraktes "Typsein" ist von der Natur her ein semantisches Konzept
  - Einfache syntaktische Konzepte können das nicht vollständig erfassen
  - Ein Universum von Typen ist selbst ein Typ, aber kein einfacher Typ
  - Ein Universum, das $\mathbb{U}$ enthält, ist ein "höheres" Universum (vgl. Menge von Mengen)
- **Führt zu Typhierarchie** $\mathbb{U} \in \mathbb{U}_2 \in \mathbb{U}_3 \ldots$
  - Typsein ist nur durch Gesamtheit aller Universen repräsentierbar
  - Komplikationen mit Mischtypen wie $S \rightarrow \mathbb{U}$ lassen sich durch kumulative Universen $\mathbb{U} \subseteq \mathbb{U}_2 \subseteq \mathbb{U}_3 \ldots$ 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**
  - $\mathbb{B}$, $\mathbb{N}$, $\times$, $\wedge$, $\vee$, $\neg$, $\exists$, $=$, ... 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 ($\rightarrow$, $\times$, $+$, $\{\}$, 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 Grundlagentheorie** (wie die Mengentheorie)
  – Semantik stützt sich "auf sich selbst" und intuitives Grundverständnis
  von (schematisierbaren) Aussagen wie "*wenn .. und .. dann*"
  – Nur die Widerspruchsfreiheit (Konsistenz) kann nachgeweisen werden

Beschreibung der Semantik logischer Aussagen

- **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 $A\ \textit{gültig} \models A \vee B\ \textit{gültig}$ 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 $A\ \textit{Aussage},\ B\ \textit{Aussage},\ \vdash A \models \ \vdash A \vee B$

Typentheorie behandelt Ausdrücke und Typen

- **Jedes Objekt hat einen Typ**
  - Schon bei der Beschreibung von Objekten wird der Typ verwendet
    Wir sagen "*Die Zahlen* $1, 2, 3, \ldots$", "*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 $1, 2, 3, \ldots$ 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** $T\ Typ$
  - 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 *(inhabited)*
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 $\mathbb{N}$ *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 $S{\rightarrow}T$ *Typ, falls* S *Typ und* T *Typ*
  - Typzugehörigkeit ebenfalls $\lambda x.t \in S{\rightarrow}T$, *falls* $t \in T$ *für alle* $x \in S$
  - Voraussetzungen für Gleichheit werden aufwendiger

    $\lambda x_1.t_1 = \lambda x_2.t_2 \in S{\rightarrow}T$, *falls* $t_1[s/x_1] = t_2[s/x_2] \in T$ *für alle* $s \in S$
- **Noch aufwendiger für abhängige Typen**
  - Gleichheit von Typausdrücken ist nicht mehr unmittelbar

    $x_1{:}S_1{\rightarrow}T_1 = x_2{:}S_2{\rightarrow}T_2$, *falls* $T_1[s/x_1] = T_2[s/x_2]$ *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)

Hypothetische Urteile

Semantische Beschreibung von Abhängigkeiten

- **Urteil wird unter Annahmen gefällt**
  - z.B. bei Gleichheit von λ-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** $\boldsymbol{x_1 \in T_1, .., x_n \in T_n[x_1..x_{n-1}] \vdash J[x_1..x_n]}$
  - 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)

Methodik zur Formulierung konkreter Typen

- **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 *Typ*, falls $T{=}T$

Eigenschaften typentheoretischer Urteile

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 $T\ \mathit{Typ}$

- **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 $T[x]\ \mathit{Typ}$ 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 $t[s_1]{=}t[s_2] \in T[s_1]$

(Neu-)Formulierung abhängiger Funktionenräume

Zentrales Konzept für Schließen über Berechnung

- **Ausdrücke**

Kanonisch:	$x:S\to T$	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**

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

- **Urteile für Typ- und Elementgleichheit**

$x_1{:}S_1\to T_1 = x_2{:}S_2\to T_2$ 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 $s_1{=}s_2 \in S_1$.

$\lambda x_1.t_1 = \lambda x_2.t_2 \in x{:}S\to T$ falls $x{:}S\to T$ *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

Definitorische Gleichheit $\equiv$

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

- **Funktionenraum** $S \to T \equiv x{:}S \to T$
  - Funktionenraum ohne Abhängigkeiten ist einfacher Spezialfall
  - T darf nicht von x abhängen
  - Alle weiteren Aspekte sind identisch
- **(Getypter) Allquantor** $\forall x{:}T.P[x] \equiv x{:}T \to P[x]$
  - 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** $A \Rightarrow B \equiv A \to B$
  - Definition stützt sich auf Einsichten der Curry-Howard Isomorphie
- **Π-Type** $\Pi_{x \in S}.T[x] \equiv x{:}S \to T[x]$
  - 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`, $\mathbb{Z}$, `Float`, $\mathbb{U}$, `{}` bereits definiert sind

Semantische Untersuchung von Eigenschaften

Zeige $\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{f}\ \mathtt{x} \in (S \to T) \to S \to T$ für beliebige S, T

Analysiere Bedingungen der semantischen Urteile

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

Semantische Untersuchung von Eigenschaften II

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

Analysiere Bedingungen der semantischen Urteile

- $\lambda\mathtt{x.x\,x}$ ist kanonisch und es gilt
  $\lambda\mathtt{x.x\,x} \in S{\rightarrow}T$ 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 $\lambda\mathtt{x.x\,x} \in S{\rightarrow}T$ 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:	match e with $\langle x\,, y\rangle \mapsto u$	x, y Variablen, e und u Terme

- **Reduktion von Ausdrücken**

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

- **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 $e.1 \;\equiv\; \mathsf{match}\ e\ \mathsf{with}\ \langle x\,,y\rangle \mapsto x$
    $e.2 \;\equiv\; \mathsf{match}\ e\ \mathsf{with}\ \langle x\,,y\rangle \mapsto y$
- **(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
- **Σ-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

Semantische Untersuchung von Eigenschaften III

Zeige $\lambda \texttt{p.(p.1)} \in (S{\times}T){\to}S$ für alle S, T

Analysiere Bedingungen der semantischen Urteile

- $\lambda \texttt{p.(p.1)}$ ist kanonisch und es gilt
  $\lambda \texttt{p.(p.1)} \in (S{\times}T){\to}S$ falls $e\texttt{.1} \in S$ für alle Terme $e \in S{\times}T$
- $e\texttt{.1}$ 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 $e \stackrel{l}{\longrightarrow} e'$, 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
  $$e.1 \stackrel{l}{\longrightarrow} (s, t).1 \stackrel{l}{\longrightarrow} s \in S$$
  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
	$\mathsf{inl}(e)$	e Term
	$\mathsf{inr}(e)$	e Term
Nichtkanonisch:	$\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(x_1) \mapsto e_1 \mid \mathsf{inr}(x_2) \mapsto e_2$	
		x_1, x_2 Variablen, e, e_1 und e_2 Terme

- **Reduktion von Ausdrücken**

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

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

- **Urteile für Typ- und Elementgleichheit**

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

Anwendungen des Summentyps

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: $\mathbb{B} \;\equiv\; \texttt{Unit+Unit}$
  - Durch Marker entstehen genau zwei unterscheidbare Elemente
    $\mathsf{T} \;\equiv\; \mathsf{inl}(),\;\; \mathsf{F} \;\equiv\; \mathsf{inr}()$
  - Nichtkanonisches Element analysiert nur den Marker
    $\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t \;\equiv\; \mathsf{case}\ b\ \mathsf{of}\ \mathsf{inl}(x) \mapsto s \mid \mathsf{inr}(y) \mapsto t$
- **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 $\lambda z.\ \texttt{case}\ z\ \texttt{of}\ \texttt{inl}(x) \mapsto x \mid \texttt{inr}(y) \mapsto (y,y)$
$\in\ z{:}S{+}T \to \texttt{case}\ z\ \texttt{of}\ \texttt{inl}(x) \mapsto S \mid \texttt{inr}(y) \mapsto T{\times}T$

Analysiere Bedingungen der semantischen Urteile

- $\lambda z.\ \texttt{case}\ z\ \texttt{of}\ \texttt{inl}(x) \mapsto x \mid \texttt{inr}(y) \mapsto (y,y)$ ist kanonisch und gehört zu $z{:}S{+}T \to \texttt{case}\ z\ \texttt{of}\ \texttt{inl}(x) \mapsto S \mid \texttt{inr}(y) \mapsto T{\times}T$
  falls $\texttt{case}\ e\ \texttt{of}\ \texttt{inl}(x) \mapsto x \mid \texttt{inr}(y) \mapsto (y,y)$
  $\in \texttt{case}\ e\ \texttt{of}\ \texttt{inl}(x) \mapsto S \mid \texttt{inr}(y) \mapsto T{\times}T$ für alle $e \in S{+}T$
- $\texttt{case}\ e\ \texttt{of}\ \texttt{inl}(x) \mapsto x \mid \texttt{inr}(y) \mapsto (y,y)$ 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 $e' = \texttt{inl}(s)$ mit $s \in S$ und $e' = \texttt{inr}(t)$ mit $t \in T$
- Aus der Annahme $e \in S{+}T$ folgt also die Existenz eines Terms s mit

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

und $\texttt{case}\ e\ \texttt{of}\ \texttt{inl}(x) \mapsto S \mid \texttt{inr}(y) \mapsto T{\times}T$

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

oder eines Terms t mit analogen Eigenschaften und somit ist

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

Logik in der Typentheorie

- **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

Logik als konservative Erweiterung

- **Definitorische Gleichheiten für Konnektive**

$$
\begin{array}{lll}
P \wedge Q & \equiv & P \times Q \\
P \vee Q & \equiv & P + Q \\
P \Rightarrow Q & \equiv & P \rightarrow Q \\
\neg P & \equiv & P \rightarrow \mathsf{f} \\
\mathsf{f} & \equiv & \{\} \\
\exists x{:}T.\,P[x] & \equiv & x{:}T \times P[x] \\
\forall x{:}T.\,P[x] & \equiv & x{:}T \rightarrow P[x]
\end{array}
$$

(Alternative: $\cap x{:}T.P[x]$ 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 "$\mathbb{U}$")
  - 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 $\mathsf{inl}(s) \in \{\}+T$ impliziert $s \in \{\}$ und $\mathsf{inr}(t) \in \{\}+T$ genau dann, wenn $t \in T$
  - $T \to \{\}$ muß leer sein, wenn T nicht leer ist (entspricht Negation)
    denn $\lambda x.a \in T \to \{\}$ impliziert $a[t/x] \in \{\}$ für alle $t \in T$
  - $\{\}\to T$ muß Elemente der Form $\lambda x.t$ für beliebige t haben,
    denn $\lambda x.t \in \{\}\to T$ falls $\{\}\to T$ *Typ* und $t[s/x] \in T$ für alle $s \in \{\}$
- **$\{\}$ braucht ein nichtkanonisches Element**
  - $\mathsf{any}(x)$ mit $\mathsf{any}(x) \in T$ für $x \in \{\}$

Keine Darstellung als konservative Erweiterung

- **Einbettung durch Definitorische Gleichheit ist möglich**

  Es gibt viele Aussagen, die prinzipiell ohne Evidenz sein müssen

  – `∀P:𝕌. P` — "*alles ist wahr*" darf nicht gelten

  – `∀X:𝕌. X=X→X in 𝕌` — Ein Typ kann nicht sein eigener Funktionenraum sein

  – `0=1 in ℕ` — in diesem Fall wären alle Zahlen gleich

  – `T=F in 𝔹` — in diesem Fall wären alle Terme gleich

  Datentypen für Universen, Gleichheit, Zahlen und 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

Der leere Datentyp präzisiert

Datentyp ohne Elemente

- **Ausdrücke**

  Kanonisch: $\{\}$

  Nichtkanonisch: any(e) e Term

- **Reduktion von Ausdrücken**

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

- **Urteile für Typ- und Elementgleichheit**

  $\{\} = \{\}$

  $e_1 = e_2 \in \{\}$ *gilt niemals*

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

Leere Datentypen ermöglichen seltsame Typen

- **$x{:}\{\}\to T$ 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 $x{:}\{\}\to T$ *Typ*, falls $\{\}$ *Typ* und $T[s]$ *Typ* für alle $s \in \{\}$

    Da $s \in \{\}$ niemals gilt, ist die Bedingung immer erfüllt also $x{:}\{\}\to T$ *Typ*

Leere Datentypen ermöglichen seltsame Typen

- **$x{:}\{\}{\to}T$ 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 $x{:}\{\}{\to}T$ *Typ*, falls $\{\}$ *Typ* und $T[s]$ *Typ* für alle $s \in \{\}$

    Da $s \in \{\}$ niemals gilt, ist die Bedingung immer erfüllt also $x{:}\{\}{\to}T$ *Typ*
- **Der Term $t = \lambda z.\,(\lambda x.\,x\,x)\,(\lambda x.\,x\,x)$ wird typisierbar**
  - Es gilt $t \in \{\}{\to}T$ für jedes beliebige T, denn

    $t \in \{\}{\to}T$ falls $\{\}{\to}T$ *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

Leere Datentypen ermöglichen seltsame Typen

- **$x{:}\{\}{\to}T$ 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 $x{:}\{\}{\to}T$ *Typ*, falls $\{\}$ *Typ* und $T[s]$ *Typ* für alle $s \in \{\}$
    Da $s \in \{\}$ niemals gilt, ist die Bedingung immer erfüllt also $x{:}\{\}{\to}T$ *Typ*
- **Der Term $t = \lambda z.\,(\lambda x.\,x\,x)\,(\lambda x.\,x\,x)$ wird typisierbar**
  - Es gilt $t \in \{\}{\to}T$ für jedes beliebige T, denn
    $t \in \{\}{\to}T$ falls $\{\}{\to}T$ *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 ($\mapsto$ §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"

$$\begin{array}{ll} \boldsymbol{P \vee_c Q} & \equiv \neg(\neg P \wedge \neg Q) \\ \boldsymbol{P \Rightarrow_c Q} & \equiv \neg P \vee_c Q \\ \boldsymbol{\exists_c x{:}T.\,P} & \equiv \neg(\forall x{:}T.\neg P) \end{array}$$

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

Grundsätzliche Eigenschaften der Theorie

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 ?

Leitlinien für Erweiterungen des Typsystems

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