Automatisierte Logik und Programmierung

Einheit 7

Abhängige Datentypen

1. Warum abhängige Datentypen?
2. Syntax, Semantik, Inferenzregeln
3. Ausdruckskraft
4. Universen und Abhängigkeit

Ziel: mehr Ausdruckskraft in der Typentheorie

- **Einfache Typisierung von $\mathbb{B}$ ist zu restriktiv**
  - Typschema für $\mathbb{B}$ ist $X \to X \to X$,
    aber in $\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t \equiv b\,s\,t$ hängt X von s und t ab
  - Einfache Typentheorie kann diese Abhängigkeit nicht beschreiben
- **Church Numerals sind Funktionen und Daten zugleich**
  - Typschema für $\mathbb{N}$ ist $(X \to X) \to X \to X$, aber
    in $\mathsf{PRs}[base, h] \equiv \lambda \mathtt{n}.\ \mathtt{n}\ h\ base$ muß $n \in \mathbb{N} = (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$ sein
  - Einfache Typentheorie kann die Doppelrolle von $\mathbb{N}$ nicht beschreiben
- **Funktionen wie $\lambda x.x$ sind polymorph**
  - Das Typschema '*für alle T gilt $\lambda x.x \in T \to T$*' ist nicht beschreibbar

Formuliere Abhängigkeit in Datentypen

- Im Ausdruck $x{:}S \to T[x]$ kann der Typ T vom Wert eines $x \in S$ abhängen
- Parameter x macht den Typ T polymorph

Informatik benötigt Abhängige Datentypen

Datenstrukturen mit internen Querbezügen

- **Abstrakte Maschinenmodelle**
  - Endliche Automaten $(Q, \Sigma, q_0, \delta, F)$ mit $q_0 \in Q$, $\delta: Q \times \Sigma \to Q$, $F \subseteq Q$
    Datentyp von q_0 hängt ab vom Wert der ersten beiden Komponenten
- **Programmiersprachliche Konstrukte**
  - Datentyp Record $[l_1 : T_1;\ \ldots;\ l_n : T_n]$
    Datentyp T_i einer Komponente hängt vom Wert des Labels l_i ab
  - Variant record `type date = Jan of 1..31 | Feb of 1..28 | ...`
    Datentyp der zweiten Komponente hängt vom Wert der ersten ab
  - Modellierung einer optionalen Eingabe abhängig von Testwerten
    `λx. if x=0 then 4 else λinput.5*input`
    Datentyp ist je nach Wert von `x` eine Zahl oder eine Funktion
- **(Programmier-)logik**
  - Quantifizierte Aussagen $(\forall x)P$, $(\exists x)P$
    Gültigkeit der Aussage P hängt vom Wert des Elements x ab

Formalisierung von Abhängigkeiten

- **Abhängiger Funktionenraum: $x{:}S \to T[x]$**
  - Elemente sind Funktionen $\lambda x.t$
  - Typ $T[x]$ des Bilds t hängt vom Wert der Eingabe $s \in S$ ab

  Einfachste Erweiterung des bisherigen Typsystems

- **Andersartige Abhängigkeiten ebenfalls möglich**
  - Abhängiges Produkt: $x{:}S \times T[x]$

    Elemente sind Paare (s, t)

    Typ der zweiten Komponente t hängt vom Wert $s \in S$ der ersten ab
  - Durchschnitt $\cap x : S \,.\, T[x]$ von Typfamilien

    Vereinigung $\cup x : S \,.\, T[x]$ von Typfamilien

    Elemente müssen zu allen/einem $T[x]$ für $x \in S$ gehören
  - Abhängiger Durchschnitt $x : S \cap T[x]$

    Element s muß zu S und zu $T[s]$ gehören (Selbstreferenz!)

    Gut für Formalisierung von abstrakten Datentypen und Objekten

$\vdots$

Abhängige Funktionenräume $x{:}S{\to}T[x]$

- **Erweiterte Syntax von Ausdrücken**
  - Typsymbole sind (atomare) Typen
  - Sind x Variable, S, T Typen, dann sind $x:S{\to}T[x]$ und (T) Typen
  - $S{\to}T$ kürzt $x:S{\to}T[x]$ ab, wenn x in T nicht frei vorkommt

  Objekt-Ausdrücke und Prioritäten sind definiert wie bisher
- **Modifizierte Typzugehörigkeitsrelation**
  - $x \in T$ — nur wenn $x \in T$ bekannt und x Termvariable
  - $\lambda y\,.\,t \in x:S{\to}T[x]$ — falls $t[x/y] \in T[x]$ aus $x \in S$ folgt
  - $f\,t \in T[t/x]$ — falls $f \in x:S{\to}T$ und $t \in S$ für einen Typ S gelten
  - $(t) \in T$, $t \in (T)$ — falls $t \in T$ gilt

  Typisierbarkeit und prinzipielles Typschema sind definiert wie bisher
- **Modifizierte Refinementregeln**
  - Abhängigkeiten der Typzugehörigkeitsrelation werden integriert

Inferenzregeln für abhängige Funktionenräume

- **Reduktions- und Deklarationsregeln**

$$\frac{H \vdash (\lambda x.u)\, s = t \in T \qquad H \vdash u[s/x] = t \in T}{}$$ `reduction`

$$\frac{H, x{:}T, H' \vdash x = x \in T}{}$$ `declaration`

- **Kongruenzregeln**

$$\frac{H \vdash f\, t = g\, u \in T \qquad H \vdash f = g \in S \to T \qquad H \vdash t = u \in S}{}$$ `applyEq` S

  – Argumenttyp S muß als Parameter der Regel angegeben werden

$$\frac{H \vdash \lambda x.t = \lambda y.u \in S \to T \qquad H, x'{:}S \vdash t[x'/x] = u[x'/y] \in T \qquad H \vdash S \in \mathbb{U}}{}$$ `lambdaEq` *(x' neu, nicht frei in H)*

- **Typdekompositionsregel**

$$\frac{H \vdash S \to T \in \mathbb{U} \qquad H \vdash S \in \mathbb{U} \qquad H \vdash T \in \mathbb{U}}{}$$ `functionI`

Inferenzregeln für abhängige Funktionenräume

- **Reduktions- und Deklarationsregeln bleiben unverändert**

$$\frac{H \vdash (\lambda x.u)\, s = t \in T \qquad H \vdash u[s/x] = t \in T}{}$$ `reduction`

$$\frac{H, x{:}T, H' \vdash x = x \in T}{}$$ `declaration`

- **Kongruenzregeln berücksichtigen Abhängigkeiten**

$$\frac{H \vdash f\, t = g\, u \in T \qquad H \vdash f = g \in x{:}\, S \to T \qquad H \vdash t = u \in S}{}$$ `applyEq` $x{:}S \to T$

  – Gesamter Typ $x{:}S \to T$ muß als Parameter der Regel angegeben werden

$$\frac{H \vdash \lambda x.t = \lambda y.u \in x{:}\, S \to T \qquad H, x'{:}S \vdash t[x'/x] = u[x'/y] \in T[x'/x] \qquad H \vdash S \in \mathbb{U}}{}$$ `lambdaEq` *(x' neu, nicht frei in H)*

- **Typdekompositionsregel berücksichtigt Abhängigkeiten**

$$\frac{H \vdash x{:}\, S \to T \in \mathbb{U} \qquad H \vdash S \in \mathbb{U} \qquad H, x'{:}S \vdash T[x'/x] \in \mathbb{U}}{}$$ `functionI`

Ausdruckskraft abhängiger Datentypen

Einfache Repräsentation vieler Basiskonzepte

- **Boolescher Ausdrücke werden polymorph formulierbar**
  - Zieldatentyp wird Parameter in Formalisierung

$$\mathsf{T} \equiv \lambda \mathtt{X}.\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x}$$

$$\mathsf{F} \equiv \lambda \mathtt{X}.\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}$$

$$\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t : X \equiv b\, X\, s\, t$$

  - Führt zur Typisierung $\mathbb{B} \equiv X : \mathbb{U} \to X \to X \to X$

    wobei $\mathbb{U}$ Repräsentant der Menge aller Datentypen
  - Typisierung von s und t liefert Wert für den Parameter X

Ausdruckskraft abhängiger Datentypen

Einfache Repräsentation vieler Basiskonzepte

- **Boolescher Ausdrücke werden polymorph formulierbar**
  - Zieldatentyp wird Parameter in Formalisierung

$$\mathsf{T} \equiv \lambda \mathtt{X}.\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x}$$

$$\mathsf{F} \equiv \lambda \mathtt{X}.\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}$$

$$\mathsf{if}\ b\ \mathsf{then}\ s\ \mathsf{else}\ t : X \equiv b\, X\, s\, t$$

  - Führt zur Typisierung $\mathbb{B} \equiv X : \mathbb{U} \to X \to X \to X$

    wobei $\mathbb{U}$ Repräsentant der Menge aller Datentypen
  - Typisierung von s und t liefert Wert für den Parameter X

- **Simulation des Summentyps verallgemeinert $\mathbb{B}$**

$$\mathtt{inl}(a) \equiv \lambda \mathtt{X}.\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x}\ a$$

$$\mathtt{inr}(b) \equiv \lambda \mathtt{X}.\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{y}\ b$$

$$\mathtt{case}\ x\ \mathtt{of}\ \mathtt{inl}(a) \to s \mid \mathtt{inr}(b) \to t : X \equiv x\, X\, (\lambda a.s)\, (\lambda b.t)$$

  - Führt zur Typisierung $A + B \equiv X : \mathbb{U} \to (A \to X) \to (B \to X) \to X$

Ausdruckskraft abhängiger Datentypen II

- **Geschlossene Typisierung von Produkten**

$$\langle u\,,v\rangle \equiv \lambda \mathtt{X}.\lambda \mathtt{p}.\ \mathtt{p}\ u\ v$$

$$\mathsf{let}\ \langle x\,,y\rangle = pair\ \mathsf{in}\ t : X \equiv pair\ X\ (\lambda x.\lambda y.t)$$

– Führt zur Typisierung $S \times T \equiv X : \mathbb{U} \to (S \to T \to X) \to X$

Ausdruckskraft abhängiger Datentypen II

- **Geschlossene Typisierung von Produkten**

$$\langle u\,,v\rangle \equiv \lambda \mathtt{X}.\lambda \mathtt{p}.\ \mathtt{p}\ u\ v$$

$$\mathsf{let}\ \langle x\,,y\rangle = pair\ \mathsf{in}\ t : X \equiv pair\ X\ (\lambda x.\lambda y.t)$$

– Führt zur Typisierung $S \times T \equiv X : \mathbb{U} \to (S \to T \to X) \to X$

- **Polymorphe Typisierung der Church-Numerals**

$$\overline{n} \equiv \lambda \mathtt{X}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\,\mathtt{x}$$

$$\mathsf{s} \equiv \lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n}(\mathbb{N})\ \mathtt{f}\ (\mathtt{f}\,\mathtt{x})$$

$$\mathsf{PRs}[base\,,h] \equiv \lambda \mathtt{n}.\ \mathtt{n}(\mathbb{N})\ h\ base$$

– Führt zu $\mathbb{N} \equiv X : \mathbb{U} \to (X \to X) \to X \to X$

– Instantiierung von X mit $\mathbb{N}$ führt zu kontrollierter Selbstreferenz

Ausdruckskraft abhängiger Datentypen II

- **Geschlossene Typisierung von Produkten**

  $\langle u, v\rangle \equiv \lambda \mathtt{X}.\lambda \mathtt{p}.\ \mathtt{p}\ u\ v$

  $\mathsf{let}\ \langle x, y\rangle = pair\ \mathsf{in}\ t : X \equiv pair\ X\ (\lambda x.\lambda y.t)$

  – Führt zur Typisierung $S \times T \equiv X : \mathbb{U} \to (S \to T \to X) \to X$

- **Polymorphe Typisierung der Church-Numerals**

  $\overline{n} \equiv \lambda \mathtt{X}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{f}^n\ \mathtt{x}$

  $\mathsf{s} \equiv \lambda \mathtt{n}.\lambda \mathtt{f}.\lambda \mathtt{x}.\ \mathtt{n}(\mathbb{N})\ \mathtt{f}\ (\mathtt{f}\ \mathtt{x})$

  $\mathsf{PRs}[base, h] \equiv \lambda \mathtt{n}.\ \mathtt{n}(\mathbb{N})\ h\ base$

  – Führt zu $\mathbb{N} \equiv X : \mathbb{U} \to (X \to X) \to X \to X$

  – Instantiierung von X mit $\mathbb{N}$ führt zu kontrollierter Selbstreferenz

- **Typisierung von Listen ist analog zu Zahlen**

  $[] \equiv \lambda \mathtt{X}.\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{x}$

  $t :: list \equiv \lambda \mathtt{X}.\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{f}\ t\ (list(\mathtt{X})\ \mathtt{f}\ \mathtt{x})$

  $\mathsf{list_ind}[b, h] \equiv \lambda \mathtt{X}.\lambda \mathtt{list}.\ \mathtt{list}(\mathtt{X})\ h\ b$

  – Führt zur Typisierung $T\ \mathrm{list} \equiv X : \mathbb{U} \to (T \to X \to X) \to X \to X$

Auch die Curry-Howard Isomorphie setzt sich fort

lambdaEq

$$\frac{H \vdash \lambda x.t \in x{:}S \to T \qquad H, x'{:}S \vdash t[x'/x] \in T[x'/x] \qquad H \vdash S \in \mathbb{U}}{}$$

allR

$$\frac{H \vdash (\forall x)B \quad \texttt{ev} = \lambda x'.\,b \qquad H, x'{:}\mathbb{U} \vdash B[x'/x] \quad \texttt{ev} = b}{}$$

applyEq $x{:}S \to T$

$$\frac{H \vdash f\,t \in T[t/x] \qquad H \vdash f \in x{:}S \to T \qquad H \vdash t \in S}{}$$

InstUniv $(\forall x)B$

$$\frac{H \vdash C[t/x] \quad \texttt{ev} = f(t) \qquad H \vdash (\forall x)B \quad \texttt{ev} = f \qquad H \vdash t \in \mathbb{U} \quad \texttt{ev} = \texttt{-}}{}$$

(Originalregel `allL` ist Variante von `InstUniv`)

Analogien zwischen Typentheorie und Logik

Typ	Formel
Variable vom Typ S	Annahme (von Evidenz für Formel) S
Term vom Typ T (freie Variablen aus S_i)	Beweis von T (Annahmen S_i)
Typechecking	Beweisführung
Abhängiger Funktionenraum	All-Quantor
Unabhängiger Funktionenraum	Implikation

Komplette Formalisierung der Logik möglich?

Logik ist durch abhängige Datentypen codierbar

- **Konjunktion**

Logik ist durch abhängige Datentypen codierbar

- **Konjunktion**
  - $A \wedge B$ gilt, wenn jede Aussage, die aus A und B folgt, gültig ist
  - $A \wedge B \equiv (\forall P)((A \Rightarrow B \Rightarrow P) \Rightarrow P)$
- **Disjunktion**

Logik ist durch abhängige Datentypen codierbar

- **Konjunktion**
  - $A \wedge B$ gilt, wenn jede Aussage, die aus A und B folgt, gültig ist
  - $A \wedge B \equiv (\forall P)((A \Rightarrow B \Rightarrow P) \Rightarrow P)$
- **Disjunktion**
  - $A \vee B$ gilt, wenn jede Aussage gültig ist, die aus A und auch aus B folgt
  - $A \vee B \equiv (\forall P)((A \Rightarrow P) \Rightarrow (B \Rightarrow P) \Rightarrow P)$
- **Negation** ist eine Abkürzung: $\neg A \equiv A \Rightarrow \mathsf{f}$

Logik ist durch abhängige Datentypen codierbar

- **Konjunktion**
  - $A \wedge B$ gilt, wenn jede Aussage, die aus A und B folgt, gültig ist
  - $A \wedge B \equiv (\forall P)((A \Rightarrow B \Rightarrow P) \Rightarrow P)$
- **Disjunktion**
  - $A \vee B$ gilt, wenn jede Aussage gültig ist, die aus A und auch aus B folgt
  - $A \vee B \equiv (\forall P)((A \Rightarrow P) \Rightarrow (B \Rightarrow P) \Rightarrow P)$
- **Negation** ist eine Abkürzung: $\neg A \equiv A \Rightarrow \mathsf{f}$
  - Ein fundamentaler Widerspruch ist die Aussage, daß jede Aussage gilt
  - $\mathsf{f} \equiv (\forall P)\, P$
- **Existenzquantor**

Logik ist durch abhängige Datentypen codierbar

- **Konjunktion**
  - $A \wedge B$ gilt, wenn jede Aussage, die aus A und B folgt, gültig ist
  - $A \wedge B \equiv (\forall P)((A \Rightarrow B \Rightarrow P) \Rightarrow P)$
- **Disjunktion**
  - $A \vee B$ gilt, wenn jede Aussage gültig ist, die aus A und auch aus B folgt
  - $A \vee B \equiv (\forall P)((A \Rightarrow P) \Rightarrow (B \Rightarrow P) \Rightarrow P)$
- **Negation** ist eine Abkürzung: $\neg A \equiv A \Rightarrow \mathsf{f}$
  - Ein fundamentaler Widerspruch ist die Aussage, daß jede Aussage gilt
  - $\mathsf{f} \equiv (\forall P)\, P$
- **Existenzquantor**
  - $(\exists x)B$ gilt, wenn jede Aussage gilt, die aus einem beliebigen $B[x]$ folgt
  - $(\exists x)\, B \equiv (\forall P)(\forall x)\,((B \Rightarrow P) \Rightarrow P)$
- **Gleichheit**

Logik ist durch abhängige Datentypen codierbar

- **Konjunktion**
  - $A \wedge B$ gilt, wenn jede Aussage, die aus A und B folgt, gültig ist
  - $A \wedge B \equiv (\forall P)((A \Rightarrow B \Rightarrow P) \Rightarrow P)$
- **Disjunktion**
  - $A \vee B$ gilt, wenn jede Aussage gültig ist, die aus A und auch aus B folgt
  - $A \vee B \equiv (\forall P)((A \Rightarrow P) \Rightarrow (B \Rightarrow P) \Rightarrow P)$
- **Negation** ist eine Abkürzung: $\neg A \equiv A \Rightarrow \mathsf{f}$
  - Ein fundamentaler Widerspruch ist die Aussage, daß jede Aussage gilt
  - $\mathsf{f} \equiv (\forall P)\, P$
- **Existenzquantor**
  - $(\exists x)B$ gilt, wenn jede Aussage gilt, die aus einem beliebigen $B[x]$ folgt
  - $(\exists x)\, B \equiv (\forall P)(\forall x)\,((B \Rightarrow P) \Rightarrow P)$
- **Gleichheit**
  - $x{=}y$ gilt, wenn kein Prädikat x von y unterscheiden kann
  - $x{=}y \equiv (\forall P)\,(P(x) \Rightarrow P(y))$

Abhängige Typen sind eine sinnvolle Erweiterung

- **Zentrale Konzepte werden darstellbar**
  - Boolesche Ausdrücke, Natürliche Zahlen, Listen, Produkte, Summen
  - Logischen Operatoren, Gleichheit
  - Abhängigkeitsstrukturen in Modellierungskonzepten (Automaten, ...)
  - Abhängigkeitsstrukturen in Programmiersprachen (Records, ...)
- **Unbegrenzte Selbstanwendung bleibt untypisierbar**
  - $\lambda x.x\ x$ ist nicht typisierbar, da $T = x{:}T{\to}T$ keine Lösung hat
  - Parametrisierung wie bei $\mathbb{N}$ führt zu stark begrenzter Selbstanwendung
    In $\lambda x.x(X)\,x$ hätte x den Typ $T = X{:}\mathbb{U}{\to}X{\to}X$
    Die Gleichung $T = T{:}\mathbb{U}{\to}T{\to}T$ enthält keine direkte Selbstreferenz
- **Es gibt einfache und natürliche semantische Modelle**
  - $x{:}S{\to}T[x]$ entspricht Abbildungen in eine Familie von Bildbereichen

Gibt es Probleme?

Abhängigkeiten und Universen

- **Abhängigkeiten benötigen "höhere Typen"**
  - Im Ausdruck $x{:}S{\to}T$ erscheint x frei in T

    Für beliebige $s \in S$ muß $T[s/x]$ Typ sein, also $T[s/x] \in \mathbb{U}$ gelten
  - Beschreibung von $x{:}S{\to}T$ benötigt Funktion $\hat{T}$ mit $\hat{T}(s) \xrightarrow{\beta} T[s/x]$

    $\hat{T}$ muß für jedes $s \in S$ einen Typ liefern, d.h. $\hat{T} \in S{\to}\mathbb{U}$
  - Der Typ von $\hat{T}$ enthält das Universum aller Typen

    Der Raum $S{\to}\mathbb{U}$ muß selbst ein Typ sein
- **Das Universum $\mathbb{U}$ bekommt eine Doppelrolle**
  - $\mathbb{U}$ ist Repräsentant des Universums aller Typen
  - $\mathbb{U}$ ist selbst ein Typ und gehört damit zum Universum aller Typen
  - Eine direkte Formalisierung dieser Einsicht wäre das Axiom $\mathbb{U} \in \mathbb{U}$

Wie tragfähig ist eine solche Typentheorie?

Die Mathematik von $\mathbb{U} \in \mathbb{U}$ (Martin-Löf, 1971)

- **Einfacher, extrem ausdrucksstarker Kalkül**
  - Ein Grundkonstrukt $x{:}S \to T$ mit 5 (bzw. 8) Inferenzregeln reicht aus
  - Alle Konzepte der Mathematik und Informatik können codiert werden

Die Mathematik von $\mathbb{U} \in \mathbb{U}$ (Martin-Löf, 1971)

- **Einfacher, extrem ausdrucksstarker Kalkül**
  - Ein Grundkonstrukt $x{:}S{\to}T$ mit 5 (bzw. 8) Inferenzregeln reicht aus
  - Alle Konzepte der Mathematik und Informatik können codiert werden
- **Russel's Paradox ist trotz $\mathbb{U} \in \mathbb{U}$ nicht formulierbar**
  - Die Menge $\{S \mid S \notin S\}$ ist nicht typisierbar
  - Beide Instanzen von S müssen verschiedene Typen haben

Die Mathematik von $\mathbb{U} \in \mathbb{U}$ (Martin-Löf, 1971)

- **Einfacher, extrem ausdrucksstarker Kalkül**
  - Ein Grundkonstrukt $x{:}S \to T$ mit 5 (bzw. 8) Inferenzregeln reicht aus
  - Alle Konzepte der Mathematik und Informatik können codiert werden
- **Russel's Paradox ist trotz $\mathbb{U} \in \mathbb{U}$ nicht formulierbar**
  - Die Menge $\{S \mid S \notin S\}$ ist nicht typisierbar
  - Beide Instanzen von S müssen verschiedene Typen haben
- **Die Theorie scheitert dennoch an Girard's Paradox**
  - Auf $\mathbb{U}$ läßt sich eine wohlfundierte Ordnung formalisieren, die eine unendliche absteigende Kette enthält (Girard, 1972)
  - Satz: **Jede Theorie mit $x{:}S \to T$ und $\mathbb{U} \in \mathbb{U}$ ist inkonsistent** (Folie 13)

Die Mathematik von $\mathbb{U} \in \mathbb{U}$ (Martin-Löf, 1971)

- **Einfacher, extrem ausdrucksstarker Kalkül**
  - Ein Grundkonstrukt $x{:}S \to T$ mit 5 (bzw. 8) Inferenzregeln reicht aus
  - Alle Konzepte der Mathematik und Informatik können codiert werden
- **Russel's Paradox ist trotz $\mathbb{U} \in \mathbb{U}$ nicht formulierbar**
  - Die Menge $\{S \mid S \notin S\}$ ist nicht typisierbar
  - Beide Instanzen von S müssen verschiedene Typen haben
- **Die Theorie scheitert dennoch an Girard's Paradox**
  - Auf $\mathbb{U}$ läßt sich eine wohlfundierte Ordnung formalisieren, die eine unendliche absteigende Kette enthält (Girard, 1972)
  - Satz: **Jede Theorie mit $x{:}S \to T$ und $\mathbb{U} \in \mathbb{U}$ ist inkonsistent** (Folie 13)
- **Andere Formalisierung des Konzepts "Typsein" nötig**
  - Konsistente und ausdruckstarke Typentheorien werden komplexer sein

Girard's Paradox

- **Grundlage: Burali-Forti Paradox für Ordinalzahlen**
  - Betrachte Menge aller Wohlordnungen $(M,<_M)$ auf Mengen $M \in \mathbb{U}$
    $<_M$ transitiv, ohne unendlich absteigende Kette $x_0 >_M x_1 >_M x_2 \ldots$
  - Definiere eine Ordnung auf dieser Menge
    $(M,<_M) <_{\mathbb{U}} (M',<_{M'})$
    $\equiv (\exists f{:}M \to M')\,(f \text{ ordnungsisomorph} \wedge (\exists y)(\forall x)(f(x) <_{M'} y))$
  - Dann ist $(\mathbb{U},<_{\mathbb{U}})$ eine Wohlordnung und $(M,<_M) <_{\mathbb{U}} (\mathbb{U},<_{\mathbb{U}})$
    gilt für jede Wohlordnung $(M,<_M)$ mit $M \in \mathbb{U}$
  - Damit ist $(\mathbb{U},<_{\mathbb{U}})$ ist ein maximales Element dieser Ordnung
  - Wegen $\mathbb{U} \in \mathbb{U}$ folgt aber auch $(\mathbb{U},<_{\mathbb{U}}) <_{\mathbb{U}} (\mathbb{U},<_{\mathbb{U}})$
  - Damit gibt es in $<_{\mathbb{U}}$ eine unendlich absteigende Kette
- **Paradox ist in allen Typtheorien mit $\mathbb{U} \in \mathbb{U}$ formulierbar**
  - Alle Ordnungskonzepte sind mit Higher-Order Quantoren definierbar
  - $x{:}S \to T$ reicht zur Formalisierung der Quantoren
  - $\mathbb{U} \in \mathbb{U}$ ist grundlegend für obiges Argument

Formalisierungen des Konzepts "Typsein"

- **Typentheorie benötigt Universum und Abhängigkeiten**
  - Universum $\mathbb{U}$ notwendig zur Formulierung des "Typseins"
  - Abhängige Datentypen sind notwendig für die Ausdruckskraft
  - Formulierung der Abhängigkeit benötigt Typen mit freien Variablen
  - $\mathbb{U}$ darf selbst kein Typ sein
- **Syntaktische Alternative** System F, Coq (Folie 15)
  - 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 16)
  - Unterschied von Objekt- und Typausdrücken liegt in der Bedeutung
    $\mathbb{U}$ und $S \to \mathbb{U}$ sind keine "einfache" Typen wie $\mathbb{N}$ oder $S \to T$
  - Universum gliedert sich in Stufen $\mathbb{U} \in \mathbb{U}_2 \in \mathbb{U}_3 \ldots$
    Nur die Gesamtheit aller Stufen beschreibt das "Typsein"

Keine der Alternativen ist perfekt

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 Spezies Variable
    und $T[x]$ ein Typ mit Parameter aus S
- **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

Semantische Auftrennung des Universums

- **Semantisch der klarste Ansatz** Martin-Löf (1978)
  - Abstraktes ‘Typsein’ ist syntaktisch nicht mit einem Konzept erfaßbar
    Ein Universum von Typen ist selbst kein einfacher Typ
  - Universum einfacher Typen ist Element eines höheren Universums
- **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 $S \to \mathbb{U}$ 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
  - Nichtsimulierbare Konzepte müssen explizit formalisiert werden
    Aufwendig für Theorieentwickler, aber nützlich für Anwender
  - Nuprl fügt weitere Typkonstrukte zu Martin-Löf’s Theorie hinzu

Typentheorie – wohin?

- **Die Theorie abhängiger Typen ist sehr ausdrucksstark**
  - Umfaßt Prädikatenlogik (Logische Struktur von Formeln)
    + λ-Kalkül (Wert von Termen)
    + Typzugehörigkeit (Eigenschaften von Termen)
- **Rahmen ist Typzugehörigkeit mit Gleichheit $s{=}t \in T$**
  - Nur die Zielsetzungen der Anwendungen sind unterschiedlich

    Logik: Finde eine Evidenz e mit $e{=}e \in [P]$ (Witness finding)

    λ-Kalkül: Berechne Wert t von s mit $s{=}t \in T$

    Type checking: Finde und prüfe Typ T mit $t{=}t \in T$ (Typ-Inferenz)

    Programmverifikation: Prüfe Eigenschaft P eines Programs, also $p{=}p \in P$

    Programmsynthese: Konstruiere Programm aus Spezifikation mit $p{=}p \in P$
  - Für alle Aspekte reicht ein einheitlicher Kalkül
- **Minimalistische Formalisierung ist inkonsistent**
  - Eine einfache Erweiterung der Theorie ist nicht möglich
- **Eine Revision der Typentheorie ist erforderlich**
  - Grundlegende Neuformulierung auf Basis der bisherigen Erkenntnisse