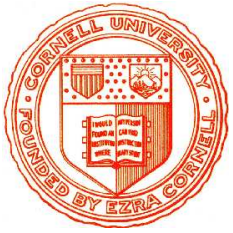
Automatisierte Logik und Programmierung

Einheit 9



Logik und Programmierung in der Typentheorie

1. Die Curry-Howard Isomorphie
2. Der leere Datentyp
3. Konstruktive und klassische Logik
4. Beweise als Programme
5. Konstruktion effizienter Algorithmen

Logik als Bibliothekskonzept

- **Logik kann in Typstruktur eingebettet werden**
  - Prinzip "*Propositions-as-types*" macht Aussagen zu Datentypen
  - Termrepräsentation eines Beweises der Aussage wird Element des Typs
- **Gleichheit ist das grundlegende Prädikat**
  - Dargestellt als Typ $s = t \in T$ mit kanonischem Element Ax
  - Mathematisch lassen sich alle Prädikate mit Gleichheiten beschreiben
- **Konnektive entsprechen Typkonstruktoren**
  - Curry-Howard Isomorphie: Inferenzregeln entsprechen einander
    - Implikation verhält sich wie (unabhängiger) Funktionenraum
    - Allquantor verhält sich wie abhängiger Funktionenraum
  - Welche Typkonstrukte ensprechen den anderen Konnektiven?
- **Logik wird zur konservativen Erweiterung**
  - Isomorphie macht "echte" Erweiterung der CTT überflüssig

Logik vs. Typentheorie: Implikation & Allquantor

$\Gamma \vdash A \Rightarrow B$

BY impI

$\Gamma, A \vdash B$

$\Gamma, A \Rightarrow B, \Delta \vdash C$

BY impE i

$\Gamma, A \Rightarrow B, \Delta \vdash A$

$\Gamma, \Delta, B \vdash C$

$\Gamma \vdash \forall x{:}T.\, A$

BY allI *

$\Gamma, x'{:}T \vdash A[x'/x]$

$\Gamma, \forall x{:}T.\, A, \Delta \vdash C$

BY allE i t

$\Gamma, \forall x{:}T.\, A, \Delta, A[t/x] \vdash C$

$\Gamma \vdash A \rightarrow B$ ⌊ext $\lambda x.\, b$⌋

BY lambdaFormation 1 THEN ...

$\Gamma,\ x{:}A \vdash B$ ⌊ext b⌋

$\Gamma,\ pf{:}\ A \rightarrow B,\ \Delta \vdash C$ ⌊ext $b[pf\ a, /y]$⌋

BY independent_functionElimination i THEN ...

$\Gamma,\ pf{:}\ A \rightarrow B,\ \Delta \vdash A$ ⌊ext a⌋

$\Gamma,\ y{:}B,\ \Delta \vdash C$ ⌊ext b⌋

$\Gamma \vdash x{:}T \rightarrow A$ ⌊ext $\lambda x'.\, a$⌋

BY lambdaFormation 1 THEN ...

$\Gamma,\ x'{:}T \vdash A[x'/x]$ ⌊ext a⌋

$\Gamma,\ pf{:}\ x{:}T \rightarrow A,\ \Delta \vdash C$ ⌊ext $u[pf\ t\ /\ y]$⌋

BY dependent_functionElimination i t

$\Gamma,\ pf{:}\ x{:}T \rightarrow A,\ \Delta \vdash t \in T$ ⌊Ax⌋

$\Gamma,\ pf{:}\ x{:}T \rightarrow A,\ y{:}A[t/x],\ \Delta \vdash C$ ⌊ext u⌋

Konjunktion & Existenzquantor

$\Gamma \vdash A \wedge B$

BY andI

$\Gamma \vdash A$

$\Gamma \vdash B$

$\Gamma \vdash A \times B$ ⌊ext $\langle a, b \rangle$⌋

BY independent_pairFormation

$\Gamma \vdash A$ ⌊ext a⌋

$\Gamma \vdash B$ ⌊ext b⌋

$\Gamma, A \wedge B, \Delta \vdash C$

BY andE i

$\Gamma, A, B, \Delta \vdash C$

$\Gamma,\ z\colon A \times B,\ \Delta \vdash C$ ⌊ext let $\langle a, b \rangle = z$ in u⌋

BY productElimination i

$\Gamma,\ z\colon A \times B,\ a\colon A,\ b\colon B,\ \Delta[\langle a, b \rangle / z] \vdash C[\langle a, b \rangle / z]$ ⌊ext u⌋

$\Gamma \vdash \exists x\colon T.\, A$

BY exI t

$\Gamma \vdash A[t/x]$

$\Gamma \vdash x\colon T \times A$ ⌊ext $\langle t, a \rangle$⌋

BY dependent_pairFormation 1 t THEN ...

$\Gamma \vdash t \in T$ ⌊Ax⌋

$\Gamma \vdash A[t/x]$ ⌊ext a⌋

$\Gamma, \exists x\colon T.\, A, \Delta \vdash C$

BY exE i **

$\Gamma, x'\colon T, A[x'/x], \Delta \vdash C$

$\Gamma,\ z\colon x\colon T \times A,\ \Delta \vdash C$ ⌊ext let $\langle x', a \rangle = z$ in u⌋

BY productElimination i THEN thin i

$\Gamma,\ x'\colon T,\ a\colon A[x'/x],\ \Delta \vdash C$ ⌊ext u⌋

Die Curry-Howard Isomorphie setzt sich fort

Typ	*Formel*
Deklaration $x{:}S$	*Annahme der Formel* S
Urteil $T[x_i]_{\lfloor \mathsf{ext}\ t \rfloor}$ *(Variablen* $x_i \in S_i$*)*	*Beweis* t *von* T *(Annahmen* S_i*)*
Funktioneraum $A \to B$	Implikation $A \Rightarrow B$
Funktionenraum $x{:}T \to A$	Universelle Quantifizierung $\forall x{:}T\,.\,A$
Produkt $A \times B$	Konjunktion $A \wedge B$
Produkt $x{:}T \times A$	Existentielle Quantifizierung $\exists x{:}T\,.\,A$
Disjunkte Vereinigung $\boxed{A+B}$	Disjunktion $A \vee B$
Leerer Datentyp $\boxed{\mathsf{Void}}$	Falschheit ff
Funktioneraum $A \to \mathsf{Void}$	Negation $\neg A$

Logik ist “umsonst”, wenn wir $A+B$ und Void haben

Disjunkte Vereinigung (Summentyp)

- **Vereinigung der Elemente zweier Typen**
  - Ursprung der Elemente von $A+B$ muß erkennbar bleiben
    - Elemente werden durch "Marker" gekennzeichnet
    - Mathematische Notation $\mathsf{inl}(a)$ bzw. $\mathsf{inr}(b)$ (linke/rechte Injektion)
  - Nichtkanonisches Element analysiert Marker und verwertet Element
    - Programmiernotation $\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}(a) \mapsto u\ \mid\ \mathsf{inr}(b) \mapsto v$
- **Boolescher Typ ist ein Spezialfall**
  - Vereinigung zweier einelementiger Typen: $\mathbb{B} \equiv \texttt{Unit+Unit}$
  - Durch Marker entstehen genau zwei unterscheidbare Elemente
  - Nichtkanonisches Element analysiert nur den Marker
- **Aufzählungstypen sind Verallgemeinerung**
  - Wichtiges Konzept vieler moderner Programmiersprachen
  - Viele Marker werden unterschieden
  - Nichtkanonisches Element entspricht komplexer Fallunterscheidung

Disjunkte Vereinigung präzisiert

Fallunterscheidung und Aufzählung von Alternativen

Syntax:

Kanonisch:	$S+T$	**union**{}(S;T)
	$\mathsf{inl}(e)$	**inl**{}(e)
	$\mathsf{inr}(e)$	**inr**{}(e)
Nichtkanonisch:	$\mathsf{case}\ \boxed{e}\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u\ \mid\ \mathsf{inr}(y) \mapsto v$	**decide**{}($\boxed{e}$; $x.u$; $y.v$)

Auswertung:

$$\mathsf{case}\ \mathsf{inl}(s)\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u\ \mid\ \mathsf{inr}(y) \mapsto v \xrightarrow{\beta} u[s/x]$$

$$\mathsf{case}\ \mathsf{inr}(t)\ \mathsf{of}\ \mathsf{inl}(x) \mapsto u\ \mid\ \mathsf{inr}(y) \mapsto v \xrightarrow{\beta} v[t/y]$$

Semantik:

$$S_1+T_1 = S_2+T_2 \equiv S_1{=}S_2 \text{ und } T_1{=}T_2$$

$$\mathsf{inl}(s_1) = \mathsf{inl}(s_2) \in S+T \equiv S+T \text{ Typ und } s_1{=}s_2 \in S$$

$$\mathsf{inr}(t_1) = \mathsf{inr}(t_2) \in S+T \equiv S+T \text{ Typ und } t_1{=}t_2 \in T$$

Inferenzregeln und Taktiken im Appendix A.3.3 des Nuprl Manuals

Logik vs. Typentheorie: Disjunktion

$\Gamma \vdash A \vee B$

BY orI1

$\Gamma \vdash A$

$\Gamma \vdash A+B$ ⌊ext inl(a)⌋

BY inlFormation j

$\Gamma \vdash A$ ⌊ext a⌋

$\Gamma \vdash B \in \mathbb{U}_j$ ⌊Ax⌋

$\Gamma \vdash A \vee B$

BY orI2

$\Gamma \vdash B$

$\Gamma \vdash A+B$ ⌊ext inr(b)⌋

BY inrFormation 1 THEN ...

$\Gamma \vdash B$ ⌊ext b⌋

$\Gamma, A \vee B, \Delta \vdash C$

BY orE i

$\Gamma, A, \Delta \vdash C$

$\Gamma, B, \Delta \vdash C$

$\Gamma,\ z{:}A+B,\ \Delta \vdash C$ ⌊ext case z of inl$(a) \mapsto u$ | inr$(b) \mapsto v$⌋

BY unionElimination i THEN thin i

$\Gamma,\ a{:}A,\ \Delta \vdash C$ ⌊ext u⌋

$\Gamma,\ b{:}B,\ \Delta \vdash C$ ⌊ext v⌋

Inferenzregeln sind im wesentlichen isomorph

Einführung eines leeren Datentyps

- **Simulierbar durch "leere Gleichheiten"**
  - $\mathsf{Void} \equiv \mathtt{X}:\mathbb{U}_1 \rightarrow \mathtt{X}=\mathtt{X}\rightarrow\mathtt{X} \in \mathbb{U}_1$
  - $\mathsf{Void} \equiv \mathtt{0}=\mathtt{1} \in \mathbb{Z}$
  - $\mathsf{Void} \equiv \mathsf{T}=\mathsf{F} \in \mathbb{B}$
- **Wichtige Eigenschaften**
  - $\mathsf{Void}\times T$ ist Datentyp ohne Elemente
  - $\mathsf{Void}+T$ ist isomorph zu T
  - $T\rightarrow\mathsf{Void}$ muß leer sein, wenn T nicht leer ist (entspricht Falschheit)
  - $\mathsf{Void}\rightarrow T$ muß das Element $\lambda x.t$ für jedes beliebige t haben
- **Void braucht nichtkanonisches Element**
  - $\mathsf{any}(x)$ mit $\mathsf{any}(x) \in T$ für $x \in \mathsf{Void}$
- **Definition als primitiver Datentyp zu bevorzugen**
  - Leere Menge ist grundlegendes mathematisches Konzept
  - Konservative Erweiterungen liefern keine nichtkanonische Elemente

Der leere Datentyp präzisiert

Gegenstück zur leeren Menge und zur logischen Falschheit

Syntax:

Kanonisch: Void **void**{}()

Nichtkanonisch: any(e) **any**{}(e)

Auswertung: —

Semantik:

Void = Void

$e = e \in$ Void *gilt niemals*

Inferenzregeln:

$\Gamma \vdash$ Void = Void $\in \mathbb{U}_j$ [Ax]
BY voidEquality

$\Gamma \vdash$ any(s) = any(t) $\in T$ [Ax]
BY anyEquality
$\Gamma \vdash s = t \in$ Void [Ax]

Γ, z: Void, $\Delta \vdash C$ [ext any(z)]
BY voidElimination i

Leere Datentypen ermöglichen seltsame Typen

In Void$\to T$ muß T kein Typ sein

– Ausdruck ist für T=λx.λy.y oder T=0=1∈2 typisierbar

```
⊢ Void→(λx.λy.y) ∈ 𝕌1
   BY independent_functionEquality
   |\
   | ⊢ Void ∈ 𝕌1
  | BY voidEquality
   \
    x:Void ⊢ λx.λy.y ∈ 𝕌1
   BY voidElimination 1
```

Gilt auch für Simulationen von Void

Void ermöglicht seltsame Typisierungen

- $\lambda z.\ (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$ **wird typisierbar**

```
⊢ ∀T:𝕌1. λz. (λx. x x) (λx. x x) ∈ Void→T
  BY allI
  \
   T:𝕌1 ⊢ λz. (λx. x x) (λx. x x) ∈ Void→T
   BY lambdaEquality 1
   |\
   | T:𝕌1, z:Void ⊢ (λx. x x) (λx. x x) ∈ T
   | BY voidElimination 2
   \
    T:𝕌1 ⊢ Void→T ∈ 𝕌1
    BY independent_functionEquality
    |\
    | T:𝕌1 ⊢ Void ∈ 𝕌2
    | BY voidEquality
    \
     T:𝕌1, x:Void ⊢ T ∈ 𝕌1
     BY hypothesisEquality 1
```

- **Gilt auch für Simulationen von** Void
- **Extrahierte Terme sind stark normalisierbar**
  - Regeln erzeugen keine Extrakt-Terme mit Selbstreferenz

Logik als Konservative Erweiterung

$P \wedge Q$	$\equiv$	$\mathsf{and}\{\}(A;B)$	$\equiv$	$P \times Q$
$P \vee Q$	$\equiv$	$\mathsf{or}\{\}(A;B)$	$\equiv$	$P + Q$
$P \Rightarrow Q$	$\equiv$	$\mathsf{implies}\{\}(A;B)$	$\equiv$	$P \rightarrow Q$
$\neg P$	$\equiv$	$\mathsf{not}\{\}(A)$	$\equiv$	$P \rightarrow \mathsf{Void}$
ff	$\equiv$	$\mathsf{false}\{\}()$	$\equiv$	Void
$\exists x{:}T\,.\,P[x]$	$\equiv$	$\mathsf{all}\{\}(T;x\,.\,A)$	$\equiv$	$x{:}T \times P[x]$
$\forall x{:}T\,.\,P[x]$	$\equiv$	$\mathsf{exists}\{\}(T;x\,.\,A)$	$\equiv$	$x{:}T \rightarrow P[x]$
$\mathbb{P}_i$	$\equiv$	$\mathsf{prop}\{i{:}\mathtt{l}\}()$	$\equiv$	$\mathbb{U}_i$

Vordefiniert in der Nuprl Bibliothek `core_1`

Konsequenzen der Curry-Howard Isomorphie

- **Eingebettete (getypte) Logik ist intuitionistisch**
  - Propositions-as-types definiert die intuitionistische Prädikatenlogik ($\mathcal{J}$)
  - Beweisregeln sind gleich, `magic` nicht simulierbar
- **Wichtige Sätze der Logik sind leicht zu zeigen**

  Schnitteliminationssatz:
  - Ist $\Gamma \vdash C$ in $\mathcal{LJ}$ beweisbar, dann gibt es einen Beweis ohne Schnittregel
    - Beweis durch Normalisierung der Extraktterme
  - $\mathcal{LJ}$ ist konsistent
    - andernfalls gäbe es einen schnittfreien Beweis für '$\vdash \mathsf{ff}$'
- **Weitere Logiken ebenfalls einbettbar**
  - Ungetypte Logik (verwende `Top` Typ als Default)
  - Prädikatenlogik beliebiger Stufe (gleiche Definition wie zuvor)
  - Funktionenkalküle höherer Ordnung

Einbettung der klassischen Logik in CTT

- **Eliminiere Konstruktivität aus Repräsentation**

ff	$\equiv$	**false**$\{\}()$	$\equiv$	Void
$\neg A$	$\equiv$	**not**$\{\}(A)$	$\equiv$	$A \rightarrow$ Void
$A \wedge B$	$\equiv$	**and**$\{\}(A;B)$	$\equiv$	$A \times B$
$\forall x{:}T\,.\,A$	$\equiv$	**all**$\{\}(T;x\,.\,A)$	$\equiv$	$x{:}T \rightarrow A$
$A \vee_c B$	$\equiv$	**orc**$\{\}(A;B)$	$\equiv$	$\neg(\neg A \wedge \neg B)$
$A \Rightarrow_c B$	$\equiv$	**impc**$\{\}(A;B)$	$\equiv$	$\neg A \vee_c B$
$\exists_c x{:}T\,.\,A$	$\equiv$	**exc**$\{\}(T;x\,.\,A)$	$\equiv$	$\neg(\forall x{:}T\,.\,\neg A)$

- **Einbettung ist "korrekt"**

  Sei $\circ$ die Einbettung der klassischen Logik und $'$ die Transformation, welche atomare Teilformel A durch $\neg\neg A$ ersetzt. Gilt $\Gamma \vdash C$ in klassischer Logik, dann gibt es einen intuitionistischen Beweis für $\Gamma^{\circ\prime} \vdash C^{\circ\prime}$

Programmierung in der Typentheorie

- **Bisherige CTT liefert nur einfache Programme**
  - λ-Terme: einfache sequentielle Funktionen
  - Paarbildung: einfachste Form einer Datenstruktur
  - Operationen auf $\mathbb{N}$: primitiv-rekursive Funktionen
  - Alle anderen Konzepte müssen durch Anwender simuliert werden
- **"Praktische" Programmierung braucht mehr**
  - "Echte" Zahlen und grundlegende arithmetische Operationen
  - Datencontainer mit flexibler Größe
  - Rekursive Daten- und Programmstrukturen

  ... unterstützt durch geeignete Inferenzmechanismen
- **Unterstütze Verifikation und Synthese**
  - Das Schreiben von Programmen ist nicht alles
  - Verifikation: nachträglicher Beweis von Programmeigenschaften
  - Synthese: Erzeuge Programme mit garantierten Eigenschaften

Arithmetik in der Typentheorie

Schlüssel für viele formale Argumente

- **Einfachste Formulierung: Peano-Aritmetik**
  - Terme $\mathbb{N}$, $\mathtt{0}$, $\mathtt{s}(t)$, $\mathtt{PR}[f,\ n, x \mapsto g](t)$ wie zuvor
- **Standardoperationen sind definierbar**
  - Konservative Erweiterung mit primitiver Rekursion

$$
\begin{aligned}
n{+}m &\equiv \mathtt{PR}[n,\ \mathtt{k},\mathtt{sum} \mapsto \mathtt{s(sum)}](m)\\
\mathbf{p}(n) &\equiv \mathtt{PR}[\mathtt{0},\ \mathtt{k},\mathtt{pre} \mapsto \mathtt{k}](n)\\
n{-}m &\equiv \mathtt{PR}[n,\ \mathtt{k},\mathtt{diff} \mapsto \mathbf{p}\mathtt{(diff)}](m)\\
n{*}m &\equiv \mathtt{PR}[\mathtt{0},\ \mathtt{k},\mathtt{prod} \mapsto n\mathtt{+prod}](m)
\end{aligned}
$$

- **Nachweis von Grundeigenschaften mühsam**
  - Die meisten Beweise benötigen aufwendige Induktionen
  - Grundeigenschaften erscheinen in Hunderten von Variationen

$$\Downarrow$$

Peano-Arithmetik nur theoretisch hinreichend

Kommutativität der Addition in Peano-Arithmetik

```
⊢ ∀n:ℕ.∀m:ℕ. n+m = m+n ∈ ℕ
  BY all_i THEN all_i
  \
   n:ℕ, m:ℕ ⊢ n+m = m+n ∈ ℕ
    BY natElimination 2
   |\
   | n:ℕ, m:ℕ ⊢ n+0 = 0+n ∈ ℕ
   | BY prReduceBase
   | \
   |  n:ℕ, m:ℕ ⊢ n = 0+n ∈ ℕ
   |  BY natElimination 1
   |  |\
   |  | n:ℕ, m:ℕ ⊢ 0 = 0+0 ∈ ℕ
   |  | BY symmetry THEN prReduceBase
   |  | \
   |  |  n:ℕ, m:ℕ ⊢ 0 = 0 ∈ ℕ
   |  |  BY zeroEquality
   |  \
   |   n:ℕ, m:ℕ, k:ℕ, f_k:k=0+k ∈ ℕ ⊢ s(k) = 0+s(k) ∈ ℕ
   |   BY symmetry THEN prReduceUp THEN symmetry
   |   \
   |    n:ℕ, m:ℕ, k:ℕ, f_k:k=0+k ∈ ℕ ⊢ s(k) = s(0+k) ∈ ℕ
   |    BY succEquality THEN hypothesis 4
   \
    n:ℕ, m:ℕ, k:ℕ, f_k:n+k=k+n ∈ ℕ ⊢ n+s(k) = s(k)+n ∈ ℕ
    BY  :
```

Zahlen in CTT: Syntax

Explizite Formalisierung der wichtigsten arithmetischen Konzepte

Kanonisch:	$\mathbb{Z}$	**int**{}()
	n	**natural_number**{n:n}()
	s<t	s<t
	$-n$	**minus**{}(**natural_number**{n:n}())
Nichtkanonisch:	$-\boxed{u}$	**minus**{}($\boxed{u}$)
	$\boxed{u}+\boxed{v}$	**add**{}($\boxed{u}$;$\boxed{v}$)
	$\boxed{u}-\boxed{v}$	**sub**{}($\boxed{u}$;$\boxed{v}$)
	$\boxed{u}*\boxed{v}$	**mul**{}($\boxed{u}$;$\boxed{v}$)
	$\boxed{u}\div\boxed{v}$	**div**{}($\boxed{u}$;$\boxed{v}$)
	$\boxed{u}$ rem $\boxed{v}$	**rem**{}($\boxed{u}$;$\boxed{v}$)
	if $\boxed{u}=\boxed{v}$ then s else t	**int_eq**($\boxed{u}$; $\boxed{v}$; s; t)
	if $\boxed{u}<\boxed{v}$ then s else t	**less**($\boxed{u}$; $\boxed{v}$; s; t)
	ind($\boxed{u}$;x,f_x.s; $base$; y,f_y.t)	**ind**{}($\boxed{u}$;x,f_x.s; $base$; y,f_y.t)

Alternative Display Form der Induktion:

rec-case u of $x<0 \mapsto [f_x]\,.\,s \mid 0 \mapsto base \mid y>0 \mapsto [f_y]\,.\,t$

Auswertung Arithmetischer Operationen

$$\mathsf{ind}(0; x, f_x.s; base; y, f_y.t) \xrightarrow{\beta} base$$

$$\mathsf{ind}(n; x, f_x.s; base; y, f_y.t) \xrightarrow{\beta} t[n, \mathsf{ind}(n\text{-}1; x, f_x.s; base; y, f_y.t)\ /y, f_y] \quad (n{>}0)$$

$$\mathsf{ind}(\text{-}n; x, f_x.s; base; y, f_y.t) \xrightarrow{\beta} s[\text{-}n, \mathsf{ind}(\text{-}n{+}1; x, f_x.s; base; y, f_y.t)\ /x, f_x] \quad (n{>}0)$$

$$\text{-}i \xrightarrow{\beta} \textit{Negation von } i \textit{ (als Zahl)}$$

$$i{+}j \xrightarrow{\beta} \textit{Summe von } i \textit{ und } j$$

$$i\text{-}j \xrightarrow{\beta} \textit{Differenz von } i \textit{ und } j$$

$$i{*}j \xrightarrow{\beta} \textit{Produkt von } i \textit{ und } j$$

$$i \div j \xrightarrow{\beta} 0, \textit{ falls } j{=}0; \textit{ sonst Integer-Division von } i \textit{ und } j$$

$$i\ \mathsf{rem}\ j \xrightarrow{\beta} 0, \textit{ falls } j{=}0; \textit{ sonst Divisionsrest von } i \textit{ und } j$$

$$\mathsf{if}\ i{=}j\ \mathsf{then}\ s\ \mathsf{else}\ t \xrightarrow{\beta} s, \textit{ falls } i = j; \textit{ ansonsten } t$$

$$\mathsf{if}\ i{<}j\ \mathsf{then}\ s\ \mathsf{else}\ t \xrightarrow{\beta} s, \textit{ falls } i < j; \textit{ ansonsten } t$$

Arithmetik: Semantik und Inferenzsystem

Semantik:

$\mathbb{Z} = \mathbb{Z}$

$s_1{<}t_1 = s_2{<}t_2 \quad \equiv \quad s_1{=}s_2{\in}\mathbb{Z}$ und $t_1{=}t_2{\in}\mathbb{Z}$

$i = i \in \mathbb{Z} \quad \equiv$ für jede Zahl i

$\mathsf{Ax} = \mathsf{Ax} \in s{<}t \quad \equiv$ Es gibt Zahlen i und j mit $s \xrightarrow{*} i$ und $t \xrightarrow{*} j$ und i ist kleiner als j

$\mathbb{Z} = \mathbb{Z} \in \mathbb{U}_j$ für jedes Level j

$s_1{<}t_1 = s_2{<}t_2 \in \mathbb{U}_j \quad \equiv \quad s_1{=}s_2{\in}\mathbb{Z}$ und $t_1{=}t_2{\in}\mathbb{Z}$, für jedes Level j

Inferenzsystem:

– 31 Inferenzregeln und zugehörige Taktiken

– Entscheidungsprozedur `arith` für elementare Arithmetik

Details im Appendix A.3.8 des Nuprl Manuals

Datenbankkonzepte:

– Große Sammlung zusätzlicher Definitionen und Theoreme

Entstanden durch Bedarf von Benutzer-Anwendungen (unvollständig)

Details in den Standard-Theorien `int_1` `int_2` `num_thy_1` der Nuprl-Bibliothek

Grundform des Datencontainers

Syntax:

Kanonisch:	T list	**list**{}(T)
	[]	**nil**{}()
	$e_1 :: e_2$	**cons**{}(e_1; e_2)
Nichtkanonisch:	list_ind($\boxed{e}$; $base$; x, l, f_{xl} . up)	**list_ind**{}($\boxed{e}$; $base$; x, l, f_{xl} . up)

Alternative Display Form der Induktion:

rec-case e of [] $\mapsto$ $base$ | $x :: l \mapsto [f_{xl}]$. up

Datenbankkonzepte:

- hd(e), tl(e), e_1@e_2, length(e), map(f;e), rev(e), $e[i]$, $e[i..j^-]$, ...
- Details in Standard Theorie list_1

Listen: Reduktion und Semantik

Auswertung:

$$\texttt{list\_ind}(\texttt{[]};\, base;\, x, l, f_{xl}.t) \xrightarrow{\beta} base$$

$$\texttt{list\_ind}(s::u;\, base;\, x, l, f_{xl}.t) \xrightarrow{\beta} t[s, u, \texttt{list\_ind}(u;\, base;\, x, l, f_{xl}.t)/x, l, f_{xl}]$$

Semantik:

$$T_1\,\mathsf{list} = T_2\,\mathsf{list} \equiv T_1{=}T_2$$

$$\texttt{[]} = \texttt{[]} \in T\,\mathsf{list} \equiv T \text{ Typ}$$

$$e_1::e_2 = e_1'::e_2' \in T\,\mathsf{list} \equiv T \text{ Typ und } e_1{=}e_1' \in T \text{ und } e_2{=}e_2' \in T\,\mathsf{list}$$

$$T_1\,\mathsf{list} = T_2\,\mathsf{list} \in \mathbb{U}_j \equiv T_1{=}T_2 \in \mathbb{U}_j$$

Inferenzregeln und Taktiken im Appendix A.3.10 des Nuprl Manuals

Wichtige benutzerdefinierte Typen

- **Logik:** $\forall$ $\exists$ $\wedge$ $\vee$ $\Rightarrow$ $\neg$ True False
  - Abbildung auf existierende Typen durch Verwendung der Curry-Howard Isomorphie (StandardTheorie `core_1`)
- **Singulärer Typ:** `Unit` $\equiv$ $0 \in \mathbb{Z}$
  - Einziges Element ist Ax (StandardTheorie `core_1`)
- **Bool'sche Logik:** $\mathbb{B}$ $\equiv$ `Unit+Unit`
  - `tt` $\equiv$ inl(Ax), `ff` $\equiv$ inr(Ax), ... (StandardTheorie `bool_1`)
  - Einbettung in Logik durch $\uparrow b$ $\equiv$ `if` b `then True else False`
- **Rekursive Funktionen:**
  - Y-Kombinator **Y** $\equiv$ `λf. (λx.f (x x)) (λx.f (x x))`
  - `letrec` $f\,x$ `=` e $\equiv$ $\mathbf{Y}(\lambda f.\lambda x.e)$

Weitere Typen definiert durch fortgeschritteneren Konzepte

Sichtweisen typentheoretischer Urteile

Wie liest man Urteile aus Sicht der Programmierung?

$T \in \mathbb{U}_j$	$t \in T$	T [ext t]
T ist eine **Menge**	t ist **Element von** T	T ist **nicht leer** *(inhabited)*
T ist **Proposition**	t ist **Beweis für** T	T ist **wahr** (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**

Lesart der Typentheorie

Logische Sicht: "Propositionen als Datentypen"

Philosophische Sichtweise (Heyting)

Konstruktive Sicht (Kolmogorov)

Beweise als Programme

- **Typentheorie ist verwendbar für Programmierung**
  - $t \in T$: "t ist Programm, das die Spezifikation T erfüllt'
  - $\vdash T\ [\mathsf{ext}\ t]$': Aufgabe, ein Programm zu konstruieren
- **Beweise für ∀-∃-Theoreme beschreiben Programme**
  - Beweis für $\vdash\ \forall \mathtt{n}:\mathbb{N}.\exists \mathtt{r}:\mathbb{N}.\ \mathtt{r}^2 \leq \mathtt{n} \wedge \mathtt{n} < (\mathtt{r}+1)^2$ liefert Extrakt-Term p_{sqrt} der Form $\lambda n.\langle r, \langle pf_1, pf_2\rangle\rangle$ mit $r = \lfloor\sqrt{n}\rfloor$
  - Quadratwurzelprogramm kann aus Beweis extrahiert werden
    - $\mathtt{sqrt} \equiv \lambda \mathtt{n}.\ \mathsf{let}\ \langle \mathtt{r}, \mathtt{pf}\rangle = p_{sqrt}(\mathtt{n})\ \mathsf{in}\ \mathtt{r}$
- **Konstruktives Auswahlaxiom formal beweisbar**

$$\vdash\ \forall \mathtt{S},\mathtt{T}:\mathbb{U}_j.\forall \mathtt{spec}:\mathtt{S}\times\mathtt{T}\to\mathbb{P}_j.$$
$$(\forall \mathtt{x}:\mathtt{S}.\ \exists \mathtt{y}:\mathtt{T}.\ \mathtt{spec}\langle \mathtt{x},\mathtt{y}\rangle)\ \Rightarrow\ \exists \mathtt{f}:\mathtt{S}\to\mathtt{T}.\forall \mathtt{x}:\mathtt{S}.\ \mathtt{spec}\langle \mathtt{x},\mathtt{f}(\mathtt{x})\rangle$$

$$\Downarrow$$

Programmentwicklung ist dasselbe wie Beweisführung

Synthese eines Quadratwurzelprogramms

- **Formaler Beweis mit Standardinduktion**

```
∀n:ℕ. ∃r:ℕ. r²≤n<(r+1)²
BY  allR
  n:ℕ  ⊢  ∃r:ℕ. r²≤n<(r+1)²
  BY  NatInd 1
  .....basecase.....
      ⊢ ∃r:ℕ. r²≤0<(r+1)²
  √  BY  existsR ⌈0⌉ THEN Auto
  .....upcase.....
      i:ℕ⁺,  r:ℕ,  r²≤i-1<(r+1)²  ⊢   ∃r:ℕ. r²≤i<(r+1)²
      BY  Decide ⌈(r+1)²≤i⌉ THEN Auto
      .....Case 1.....
          i:ℕ⁺,  r:ℕ,  r²≤i-1<(r+1)²,  (r+1)²≤i
          ⊢ ∃r:ℕ. r²≤i<(r+1)²
      √  BY  existsR ⌈r+1⌉ THEN Auto'
      .....Case 2.....
          i:ℕ⁺,  r:ℕ,  r²≤i-1<(r+1)²,  ¬((r+1)²≤i)
          ⊢ ∃r:ℕ. r²≤i<(r+1)²
      √  BY  existsR ⌈r⌉ THEN Auto
```

- **Extrahierter linearer Algorithmus (nach Projektion)**

```
let rec sqrt n = if n=0 then 0
                 else let r = sqrt (n-1)  in
                 if (r+1)²≤n  then r+1 else r
```

Erzeugung eines effizienteren Algorithmus

- **Formaler Beweis mit Binärinduktion**

```
∀n:ℕ. ∃r:ℕ. r²≤n<(r+1)²
BY  allR
  n:ℕ  ⊢  ∃r:ℕ. r²≤n<(r+1)²
  BY  NatInd4 1
  .....basecase.....
      ⊢ ∃r:ℕ. r²≤0<(r+1)²
  √  BY  existsR ⌈0⌉ THEN Auto
  .....upcase.....
      i:ℕ,  r:ℕ,  r²≤i÷4<(r+1)²  ⊢  ∃r:ℕ. r²≤i<(r+1)²
      BY  Decide ⌈((2*r)+1)²≤i⌉ THEN Auto
      .....Case 1.....
          i:ℕ,  r:ℕ,  r²≤i÷4<(r+1)²,  ((2*r)+1)²≤i
          ⊢ ∃r:ℕ. r²≤i<(r+1)²
      √  BY  existsR ⌈(2*r)+1⌉ THEN Auto'
      .....Case 2.....
          i:ℕ,  r:ℕ,  r²≤i÷4<(r+1)²,  ¬(((2*r)+1)²≤i)
          ⊢ ∃r:ℕ. r²≤i<(r+1)²
      √  BY  existsR ⌈2*r⌉ THEN Auto
```

- **Extrahierter Algorithmus hat logarithmische Laufzeit**

```
let rec sqrt n = if n=0 then 0
                 else let r = sqrt (n/4)  in
                 if (2*r+1)²≤n  then 2*r+1 else 2*r
```