Formale Techniken der Software-Entwicklung

Matthias Hölzl, Christian Kroiß

16. Juni 2014

Normalformen der Prädikatenlogik

Definition

Eine prädikatenlogische Formel ϕ ist genau dann in *Pränex-Normalform*, wenn sie die Form

$$Q_1\, x_1 \ldots . Q_n\, x_n . \mu$$

hat, wobei $Q_i \in \{\forall, \exists\}$ und μ eine quantorenfreie Formel ist.
$Q_1\, x_1 \ldots . Q_n\, x_n$ heißt der *Präfix*, μ die *Matrix* der Formel.

Zu jeder Formel gibt es (mindestens) eine Pränex-Normalform.

Wenn eine Formel ϕ in Pränex-Form ist, ihr Präfix nur aus Allquantoren besteht, und ihre Matrix in CNF ist, dann sagt man ϕ sei in *Skolem-Normalform* oder *Standardform*. Üblicherweise lässt man den Präfix dann weg.

Skolem-Normalform erhält Inkonsistenz

Proposition

Sei ϕ eine Formel und S die nach dem oben beschriebenen Verfahren erhaltene Skolem-Normalform zu ϕ. S ist genau dann inkonsistent, wenn ϕ inkonsistent ist.

Die Skolem-Normalform ϕ_S einer Formel ϕ ist *nicht* äquivalent zu ϕ. Sei, zum Beispiel

$$\phi = \exists x.P(x)$$

und $\mathcal{A}$ die folgende Struktur

Träger	$\{1, 2\}$
Konstanten	$c^{\mathcal{A}} = 1$
Prädikate	$P(1)^{\mathcal{A}} = \text{falsch}$, $P(2)^{\mathcal{A}} = \text{wahr}$

Dann gilt $\mathcal{A} \models \phi$ aber $\mathcal{A} \not\models \phi_S$

Unifikatoren

Definition (Komposition von Substitutionen)

Seien σ und θ Substitutionen. Die Komposition von θ und σ, $\theta \circ \sigma$ schreibt man auch $\sigma\theta$.

Bei der Komposition von Substitutionen wird also die links stehende Substitution zuerst ausgeführt. Das ist sinnvoll, weil die a Substitution von rechts angewendet wird: $(\phi\sigma)\theta = \phi(\sigma\theta) = \phi(\theta \circ \sigma)$.

Definition

Sei $E = \{E_1, \ldots E_k\}$ eine Menge von Ausdrücken (Termen oder Formeln). Eine Substitution σ heißt *Unifikator* von E wenn $E_1\sigma \equiv E_2\sigma \equiv \ldots \equiv E_k\sigma$ ist. E heißt *unifizierbar* wenn es einen Unifikator von E gibt. Ein Unifikator σ heißt *allgemeinster Unifikator* (*most general Unifyer*, *MGU*) von E, wenn sich jeder andere Unifikator θ in der Form $\theta = \sigma\eta$ schreiben lässt.

Binäre Resolution

Seien $L = \{L_1, \ldots, L_m\}$ und $M = \{M_1, \ldots, M_n\}$ so, dass L und M keine gemeinsamen Variablen haben. Falls L_i und M_j mit MGU σ unifizierbar sind und $L_i\sigma$ und $M_j\sigma$ komplementär sind (d.h. $L_i\sigma \equiv \neg M_j\sigma$ oder $\neg L_i\sigma \equiv M_j\sigma$) gilt

$$\frac{L_1, \ldots, L_m \qquad M_1, \ldots, M_n}{(L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m, M_1 \ldots, M_{j-1}, M_{j+1}, M_n)\sigma}$$

Diese Form der Ableitung heißt binäre Resolution.
$(L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m, M_1 \ldots, M_{j-1}, M_{j+1}, \ldots, M_n)\sigma$ heißt *binäre Resolvente* von $L_1, \ldots, L_m$ und $M_1, \ldots, M_n$.

Binäre Resolution

Binäre Resolution für Prädikatenlogik ist keine vollständige Ableitungsregel. Man muss binäre Resolution entweder durch Resolution von Mengen unifizierbarer komplementärer Literale erweitern oder sog. Faktorisierung einführen.

Definition

Sei $L = \{L_1, \ldots L_m\}$ eine Menge von Literalen. Sind zwei oder mehrere L_i mit MGU σ unifizierbar, so heißt $L\sigma$ ein *Faktor* von L.

Resolution

Seien L und M Mengen von Literalen. N ist eine Resolvente von L und M, wenn die Resolvente einer Anwendung binärer Resolution auf

1. L und M,
2. L und einen Faktor von M,
3. einen Faktor von L und M oder
4. einen Faktor von L und einen Faktor von M

ist. Die Ableitung einer solchen Resolvente nennt man *(prädikatenlogische) Resolution*. Wir schreiben dann $L, M \left|\frac{}{\text{res}}\right. M$.

Vollständigkeit der Resolution

Wir schreiben $S \vdash_{\text{res}} M$, wenn durch prädikatenlogische Resolution aus $S, \neg M$ die leere Klausel abgeleitet werden kann. Dann gilt:

Satz

Prädikatenlogische Resolution ist vollständig und korrekt, d.h., es gilt

$$S \vdash_{\text{res}} M \quad \textit{genau dann, wenn} \quad S \models M$$

Der Beweis erfolgt durch den Satz von Herbrand bzw. die im Satz von Herbrand verwendete Konstruktion.

(Wie im aussagenlogischen Fall kann durch den prädikatenlogischen Resolutionskalkül nicht direkt M aus S abgeleitet werden, wenn $S \models M$ gilt.)

Satz von Herbrand

Satz (Herbrand)

Eine Menge von Klauseln S ist genau dann unerfüllbar, wenn es eine endliche, unerfüllbare Menge S' gibt, die aus Grundinstanzen von S besteht.

Der Satz von Herbrand ist die Rechtfertigung für automatisierte Beweisverfahren wie den DPLL-Algorithmus und die Grundlage für den Korrektheitsbeweis des Resolutionskalküls.

Korrektheit und Vollständigkeit

Sei Φ eine (möglicherweise unendliche) Menge von Formeln. Dann bedeuten

- Korrektheit eines Beweissystems

$$\Phi \vdash \phi \quad \text{impliziert} \quad \Phi \models \phi$$

- Vollständigkeit eines Beweissystems

$$\Phi \models \phi \quad \text{impliziert} \quad \Phi \vdash \phi$$

Gödelscher Vollständigkeitssatz

Theorem (Gödelscher Vollständigkeitssatz)

Sei $\Phi \subseteq \mathcal{L}$ eine Menge von Formeln und $\phi \in \mathcal{L}$ eine Formel. Dann gilt

$$\Phi \vdash_{\text{seq}} \phi \quad \textit{genau dann, wenn} \quad \Phi \models \phi$$

$$\Phi \vdash_{\text{res}} \phi \quad \textit{genau dann, wenn} \quad \Phi \models \phi$$

Endlichkeits-, Kompaktheitssatz

Wir haben gezeigt, dass aus dem Gödelschen Vollständigkeitssatz und dem Endlichkeitssatz unmittelbar folgt:

Theorem (Endlichkeitssatz für das Folgern)

Wenn $\Phi \models \phi$ gilt, so gibt es eine endliche Teilmenge Φ_0 für die $\Phi_0 \models \phi$ gilt.

Ähnlich zeigt man die folgende Aussage

Satz (Kompaktheitssatz)

Eine Formelmenge Φ ist genau dann erfüllbar, wenn jede endliche Teilmenge von Φ erfüllbar ist.

Ausdrucksmächtigkeit der Prädikatenlogik

Viele Sachverhalte können in der Prädikatenlogik ausgedrückt werden:

- Mathematische Theorien, z.B. Gruppentheorie, Mengenlehre, ...
- Chemische Reaktionen
- Abläufe (durch Formalisierung von Situationen, Fluenten, etc.)

Aber: Manche Sachverhalte lassen sich nicht beschreiben, z.B. die transitive Hülle: Wir haben das Prädikat E für Eltern definiert. Das Prädikat V soll die Vorfahren repräsentieren, $V(x, y)$ genau dann, wenn y ein Vorfahre von x ist.

- $P(x, y) \Rightarrow V(x, y)$
- $(\exists z. P(x, z) \wedge V(z, y)) \Rightarrow V(x, y)$

Ist V eine korrekte Formalisierung des Begriffs „Vorfahre“?

Transitiver Abschluss

Frage: Ist V eine korrekte Formalisierung des Begriffs „Vorfahre“?

Antwort: Nein. V erzwingt zwar die Transitivität, stellt aber nicht sicher, dass das Modell von V minimal ist. Z.B. kann man V in einer Struktur $\mathcal{A}$ durch A^2 interpretieren. Was wir suchen ist die minimale Relation, für die die geforderte Abschlusseigenschaft gilt; *nur* die Elemente, die durch die Axiome gefordert sind sollen in V enthalten sein.

Frage: Wie können wir die Minimalität von V in der Prädikatenlogik ausdrücken?

Antwort: Gar nicht. Die Minimalität von V ist eine Eigenschaft, die über alle möglichen Prädikate redet („V impliziert *jedes andere Prädikat*, das . . . “). Das ist in der Prädikatenlogik nicht ausdrückbar.

Prädikatenlogik erster Stufe

In der Prädikatenlogik (erster Stufe) können wir unterscheiden, ob eine Aussage nur für bestimmte Objekte wahr ist, oder für alle möglichen Objekte:

$$1 \cdot x = x$$

ist nur für die Konstante 1, aber für jeden möglichen Wert der Variablen x wahr. Die Aussage

$$P(x) = P(x)$$

ist für alle x wahr und äquivalent zu

$$\forall x.P(x) = P(x)$$

Sie ist aber auch für jede beliebige Wahl von P wahr, aber da die Prädikatenlogik für Prädikate und Funktionen nur Konstanten anbietet, kann man das nicht ausdrücken.

Prädikatenlogik zweiter Stufe

Erweitert man die Prädikatenlogik um Variablen für Funktionen und Prädikate und erlaubt Quantifikation über diese Variablen, so erhält man die *Prädikatenlogik zweiter Stufe*. Damit kann man z.B. schreiben

$$\forall P.\forall x.P(x) = P(x)$$

Mit dieser Erweiterung kann man die Transitive Hülle V von P folgendermaßen charakterisieren:

$$\forall Q.\Big[P(x,y) \Rightarrow Q(x,y) \wedge \big((\exists z.P(x,z) \wedge Q(z,y)) \Rightarrow Q(x,y)\big)\Big] \Rightarrow V(x,y) \Rightarrow Q(x,y)$$

Prädikatenlogik zweiter Stufe

Als Argumente von Funktionen und Prädikaten sind in der Prädikatenlogik zweiter Stufe nur Individuen zulässig, keine Funktions- oder Prädikatenvariablen oder -konstanten. Z.B. ist $P(f)$ kein zulässiger Term, wenn f eine Funktionsvariable ist.

Syntax: Terme

Seien Var_I eine Menge von Individuenvariablen, Var_F eine Menge von Funktionsvariablen, Fun eine Menge von Funktionssymbolen und $Const$ eine Menge von Konstantensymbolen. Sei $|.| : Var_F \cup Fun \to \mathbb{N}$ eine Abbildung, die jeder Funktionsvariable und jedem Funktionssymbol eine *Stelligkeit* zuordnet. Ist $|f| = n$, so nennt man f n-stellig.
Die Menge der prädikatenlogischen Terme $\mathcal{T}$ (über Var_I, Var_F, Fun und $Const$) ist folgendermaßen rekursiv definiert.

- Jede Variable $x \in Var$ ist in $\mathcal{T}$ (d.h. $Var_I \subseteq \mathcal{T}$)
- Jede Konstante $c \in Const$ ist in $\mathcal{T}$ (d.h. $Const \subseteq \mathcal{T}$)
- Sei f eine n-stellige Funktionsvariable oder ein n-stelliges Funktionssymbol und seien $t_1, \ldots, t_n$ Terme, dann ist $f(t_1, \ldots, t_n) \in \mathcal{T}$

Variablen und Konstanten nennt man auch *Primterme* oder *atomare Terme* oder *Atome*. Alle anderen Terme heißen *Funktionsterme*.

Syntax: Atomare Formeln

Sei $\mathcal{T}$ eine Menge von Termen (über Var_I, Var_F, *Fun* und *Const*), Var_P eine Menge von Prädikatenvariablen und *Pred* eine Menge von Prädikatensymbolen. Sei $|.| : Var_P \cup Pred \to \mathbb{N}$ eine Abbildung, die jeder Prädikatenvariablen und jedem Prädikatensymbol eine *Stelligkeit* zuordnet. Ist $|P| = n$, so nennt man P ein n-stelliges Prädikat.

Die *atomaren Formeln* (*Primformeln*) $\mathcal{A}$ (über *Var*, *Fun*, *Const* und *Pred*) sind dann folgendermaßen rekursiv definiert:

- Seien P eine n-stellige Prädikatenvariable oder ein n-stelliges Prädikatensymbol und $t_1, \ldots, t_n \in T$. Dann ist $P(t_1, \ldots, t_n) \in \mathcal{A}$
- Für $t_1, t_2 \in \mathcal{T}$ ist $t_1 = t_2 \in \mathcal{A}$

Syntax: Formeln

Seien Var_I, Var_F, Var_P Mengen von Individuen, Funktions- und Prädikatenvariablen und $\mathcal{A}$ die Menge von atomaren Formeln über *Var*, *Fun*, *Const* und *Pred*. Die Menge der prädikatenlogischen Formeln $\mathcal{L}$ (über *Var*, *Fun*, *Const* und *Pred*) ist dann folgendermaßen rekursiv definiert:

- Ist $\phi \in \mathcal{A}$ so ist $\phi \in \mathcal{L}$ (d.h. $\mathcal{A} \subseteq \mathcal{L}$)
- Sind $\phi, \psi \in \mathcal{L}$, so sind auch $\neg\phi$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \Rightarrow \psi)$ und $(\phi \Leftrightarrow \psi)$ in $\mathcal{L}$ enthalten
- Sind $x \in Var_I$ und $\phi \in \mathcal{L}$, so sind auch $\forall x.\phi$ und $\exists x.\phi$ in $\mathcal{L}$ enthalten
- Sind $f \in Var_F$ und $\phi \in \mathcal{L}$, so sind auch $\forall f.\phi$ und $\exists f.\phi$ in $\mathcal{L}$ enthalten
- Sind $P \in Var_P$ und $\phi \in \mathcal{L}$, so sind auch $\forall P.\phi$ und $\exists P.\phi$ in $\mathcal{L}$ enthalten

Formeln nennt man auch Ausdrücke oder Aussageformen.

Semantik der Prädikatenlogik zweiter Stufe

Die Semantik von Termen der Prädikatenlogik zweiter Stufe wird genau wie für die Prädikatenlogik erster Stufe definiert, nur muss jetzt auch Termen mit Quantoren über Funktionen und Prädikaten eine Bedeutung gegeben werden. Wie das geht ist aber recht nahe liegend.

Strukturen

Strukturen werden genau wie für die Prädikatenlogik definiert: Eine Struktur $\mathcal{A}$ zur Signatur $\sigma = (\mathit{Const}, \mathit{Fun}, \mathit{Pred}, |.|)$ (kurz σ-Struktur) besteht aus

- einer nichtleeren Menge A, dem *Träger* (oder der *Grundmenge*) der Struktur
- einem Element $c^{\mathcal{A}} \in A$ für jedes Konstantensymbol $c \in \mathit{Const}$
- einer Funktion $f^{\mathcal{A}} : A^n \to A$ für jedes n-stellige Funktionssymbol $f \in \mathit{Fun}$
- einer Relation $P^{\mathcal{A}} \subseteq A^n$ für jedes n-stellige Prädikatensymbol P

Modelle der Prädikatenlogik zweiter Stufe

Definition

Ein *Modell* $\mathcal{M}$ einer prädikatenlogischen Sprache zweiter Stufe $\mathcal{L}$ ist ein Paar $(\mathcal{A}, w)$, bestehend aus einer $\mathcal{L}$-Struktur $\mathcal{A}$ (mit Träger A) und *Belegungen* $w_I : Var_I \to A$, $w_F : Var_F \to \mathcal{F}(A)$, $w_P : Var_P \to \mathcal{R}(A)$. Wir schreiben $[\![c]\!]_{\mathcal{M}}$ (oder $c^{\mathcal{M}}$, oder $[\![c]\!]$ falls $\mathcal{M}$ klar ist) für $c^{\mathcal{A}}$, entsprechend für Funktions- und Prädikatensymbole.

Dabei steht $\mathcal{F}(A)$ für die Menge aller Funktionen über A und $\mathcal{R}(A)$ für die Menge aller Relationen über A; die Belegungen w_F und w_P müssen die Stelligkeiten der Funktions- bzw. Prädikatenvariablen berücksichtigen.

Substitution in Modellen wird sinngemäß wie bei der Prädikatenlogik erster Stufe definiert.

Semantik von Termen

Durch ein Modell $\mathcal{M}$ wird jedem $\mathcal{L}$-Term ein Element aus A zugeordnet:

$$
\begin{aligned}
[\![x]\!]_{\mathcal{M}} &= w_I(x) \\
[\![c]\!]_{\mathcal{M}} &= c^{\mathcal{A}} \\
[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{M}} &= w_F(f)([\![t_1]\!]_{\mathcal{M}}, \ldots, [\![t_n]\!]_{\mathcal{M}}) && f \text{ Funktionsvariable} \\
[\![f(t_1, \ldots, t_n)]\!]_{\mathcal{M}} &= f^{\mathcal{A}}([\![t_1]\!]_{\mathcal{M}}, \ldots, [\![t_n]\!]_{\mathcal{M}}) && f \text{ Funktionssymbol}
\end{aligned}
$$

Erfüllungsrelation

Die Semantik von Formeln lässt sich durch die *Erfüllungsrelation* $\mathcal{M} \models \phi$ ($\mathcal{M}$ erfüllt ϕ oder $\mathcal{M}$ ist ein Modell von ϕ) beschreiben:

$$\mathcal{M} \models P(t_1, \ldots, t_n) \iff w_P(P)([\![t_1]\!]_{\mathcal{M}}, \ldots, [\![t_n]\!]_{\mathcal{M}}) \quad P \text{ Prädikatenvariable}$$

$$\mathcal{M} \models P(t_1, \ldots, t_n) \iff P^{\mathcal{A}}([\![t_1]\!]_{\mathcal{M}}, \ldots, [\![t_n]\!]_{\mathcal{M}}) \quad P \text{ Prädikatensymbol}$$

$$\mathcal{M} \models t_1 = t_2 \iff [\![t_1]\!]_{\mathcal{M}} = [\![t_2]\!]_{\mathcal{M}}$$

$$\mathcal{M} \models \neg\phi \iff \mathcal{M} \models \phi \text{ ist falsch } (\mathcal{M} \not\models \phi)$$

$$\mathcal{M} \models \phi \wedge \psi \iff \mathcal{M} \models \phi \text{ und } \mathcal{M} \models \psi$$

$$\mathcal{M} \models \phi \vee \psi \iff \mathcal{M} \models \phi \text{ oder } \mathcal{M} \models \psi$$

$$\mathcal{M} \models \phi \Rightarrow \psi \iff \mathcal{M} \not\models \phi \text{ oder } \mathcal{M} \models \psi$$

$$\mathcal{M} \models \phi \Leftrightarrow \psi \iff (\mathcal{M} \models \phi \text{ und } \mathcal{M} \models \psi) \text{ oder } (\mathcal{M} \not\models \phi \text{ und } \mathcal{M} \not\models \psi)$$

Erfüllungsrelation

Die Semantik von quantifizierten Formeln wird folgendermaßen definiert:

$$\mathcal{M} \models \forall x.\phi \iff \mathcal{M}[x \mapsto a] \models \phi \text{ für alle Elemente } a \in A$$
$$\mathcal{M} \models \exists x.\phi \iff \text{es gibt ein Element } a \in A \text{ mit } \mathcal{M}[x \mapsto a] \models \phi$$
$$\mathcal{M} \models \forall f.\phi \iff \mathcal{M}[f \mapsto g] \models \phi \text{ für alle Funktionen } g \in \mathcal{F}(A)$$
$$\mathcal{M} \models \exists f.\phi \iff \text{es gibt eine Funktion } g \in \mathcal{F}(A) \text{ mit } \mathcal{M}[f \mapsto g] \models \phi$$
$$\mathcal{M} \models \forall P.\phi \iff \mathcal{M}[P \mapsto R] \models \phi \text{ für alle Relationen } R \in \mathcal{R}(A)$$
$$\mathcal{M} \models \exists x.\phi \iff \text{es gibt eine Relation } R \in \mathcal{R}(A) \text{ mit } \mathcal{M}[P \mapsto R] \models \phi$$

Auch hier müssen die Funktionen und Relationen die Stelligkeit beachten.

Ausdrucksmächtigkeit der Prädikatenlogik zweiter Stufe

- Endlichkeit des Modells:

$$\forall f.(f(x) = f(y) \Rightarrow x = y) \Rightarrow \forall y.\exists x.f(x) = y$$

  (Jede injektive Funktion ist surjektiv.)
- Unendlichkeit des Modells:

$$\exists R.(\forall x.\forall y.\forall z.R(x,y) \wedge R(y,y) \Rightarrow R(x,z)) \wedge \forall x.\neg R(x,x) \wedge \exists y.R(x,y)$$

  (Es gibt eine vollständige, transitive, irreflexive Relation.)
- Wohlordnung (wenn eine Prädikat erfüllbar ist, so gibt es ein kleinstes Element, welches das Prädikat erfüllt):

$$\forall P.(\exists x.P(x)) \Rightarrow \exists x.P(x) \wedge \forall y.P(y) \Rightarrow x \leq y$$

- Transitive Hülle (s.o.)
- „There are some critics who admire only each other“

Eigenschaften der Prädikatenlogik zweiter Stufe

Sei $\phi_{\geq n}$ die Formel

$$\exists x_1 \ldots \exists x_n . x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge \cdots \wedge x_{n-1} \neq x_n$$

die besagt, dass es mindestens n verschiedene Individuen gibt, sei

$$\phi_{endl} = \forall f . (f(x) = f(y) \Rightarrow x = y) \Rightarrow \forall y . \exists x . f(x) = y$$

die Formel, die besagt, dass alle Modelle endlich sein müssen, und sei

$$\Phi = \{\phi_{endl}\} \cup \{\phi_{\geq n} \mid n \in \mathbb{N}\}$$

Dann ist jede endliche Teilmenge von Φ erfüllbar aber Φ nicht (denn kein endliches Modell kann $\phi_{\geq n}$ für alle n erfüllen, kein unendliches Modell ϕ_{endl}.) Damit ist klar, dass der Kompaktheitssatz nicht gelten kann.

Eigenschaften der Prädikatenlogik zweiter Stufe

- Der Kompaktheitssatz gilt nicht
- Da der Kompaktheitssatz unmittelbar aus dem Vollständigkeitssatz und den Eigenschaften von Herleitungen folgt, kann der Vollständigkeitssatz auch nicht gelten: Kein korrekter Beweiskalkül für die Prädikatenlogik zweiter Stufe ist vollständig
- Die Menge der allgemeingültigen Formeln zweiter Stufe ist nicht rekursiv aufzählbar
- Der Satz von Löwenheim-Skolem gilt nicht für die Prädikatenlogik zweiter Stufe

Prädikatenlogiken höherer Stufe

Frage: Was passiert, wenn man Funktionen und Prädikate als Argumente von Funktionen und Prädikaten zulässt? Bekommt man immer ausdrucksstärkere Logiken?

Antwort: Nein. Man kann zeigen, dass Logiken höherer Stufe keine weitere Steigerung der Ausdrucksmächtigkeit bringen. Sie lassen sich alle auf die Prädikatenlogik zweiter Stufe zurückführen.

Für die praktische Anwendung ist es aber vorteilhaft Terme und Formeln höherer Ordnung zuzulassen und zu systematisieren. Damit kommt man zur sog. (klassischen) *Typtheorie*.

Prädikate und Funktionen

Prädikate repräsentieren Relationen. Eine Relation kann man aber mit ihrer charakteristischen Funktion identifizieren. Wenn man also Boole'sche Funktionen hat, braucht man Relationen nicht mehr als eigenständige Objekte. Daher führen wir die Typtheorie in einer funktionalen Form ein, in der wir syntaktisch nicht mehr zwischen Termen und Formeln unterscheiden.

Typtheorie

Wir betrachten ein Typsystem, das zwei primitive Typen und einen Typkonstruktor hat:

- i ist der Typ der Individuen
- o ist der Typ von Aussagen
- $\to$ ist ein zweistelliger Typkonstruktor, der nach rechts assoziiert

Damit können wir einstellige Prädikate durch den Typ $i \to o$ und einstellige Funktionen durch den Typ $i \to i$ repräsentieren. Mehrstellige Funktionen und Relationen werden wie in der funktionalen Programmierung „gecurried“ dargestellt: eine zweistellige Funktion erhält den Typ $i \to (i \to i) = i \to i \to i$. Eine Funktion zweiter Ordnung, die eine Funktion erster Ordnung als Argument erhält, bekommt den Typ $(i \to i) \to i$.

Terme/Formeln

Die Typtheorie hat die folgenden Symbole:

- Für jeden Typ α gibt es eine abzählbare Menge von Variablen x_α
- Die logischen Konstanten $\neg_{o\to o}$, $\vee_{o\to o\to o}$, $\Pi_{(\alpha\to o)\to o}$ und $\iota_{(\alpha\to o)\to\alpha}$, für jeden Typ α
- Nichtlogische (domänenspezifische) Konstanten

Terme/Formeln sind folgendermaßen definiert:

- Jede Variable und Konstante vom Typ α ist ein Term vom Typ α
- Sind $A_{\alpha\to\beta}$ und B_α Terme, so ist AB ein Term vom Typ β
- Ist x_α eine Variable und A_β ein Term so ist $\lambda x.A$ ein Term vom Typ $\alpha \to \beta$

Abkürzungen

Man schreibt

- $A_o \vee B_o$ für $\vee_{o\to o\to o} AB$
- $A_o \Rightarrow B_o$ für $(\neg_{o\to o} A_o) \vee B_o$
- $\forall x_\alpha . A_o$ für $\Pi_{\alpha\to o}(\lambda x_\alpha . A_o)$
- ...

Ableitungsregeln

- Umbenennung gebundener Variablen
- β-Reduktion: $(\lambda x.A)B \rightsquigarrow A[x \mapsto B]$ (falls B substituierbar für x in A)
- β-Expansion: Umkehrung von β-Reduktion
- Substitution: $F_{\alpha\to o}x_\alpha \rightsquigarrow F_{\alpha\to o}A_\alpha$ falls x nicht frei in F vorkommt
- Modus Ponens: aus $A \Rightarrow B$ und A folgt B
- Generalisierung: aus $F_{\alpha\to o}x_\alpha$ folgt $\Pi_{(\alpha\to o)\to o}F_{\alpha\to o}$

Die ersten beiden Ableitungsregeln sind die aus dem (ungetypten) Lambda-Kalkül bekannten η-Konversions- und β-Reduktionsregeln. Damit lassen sich manche Terme in diesem Kalkül als funktionale Programme auffassen.

Axiome

Die folgenden Axiome können in Ableitungen benutzt werden:

- $(x_o \vee x_o) \Rightarrow x_o$
- $x_o \Rightarrow (x_o \vee y_o)$
- $x_o \vee y_o \Rightarrow y_o \vee x_o$
- $x_o \Rightarrow y_o \Rightarrow \big((z_o \vee x_o) \Rightarrow (z_o \vee y_o)\big)$
- $\Pi_{(\alpha \to o) \to o} f_{\alpha \to o} \Rightarrow f_{\alpha \to o} x_\alpha$
- $(\forall x_\alpha . y_o \vee f_{\alpha \to o} x_\alpha) \Rightarrow y_o \vee \Pi_{(\alpha \to o) \to o} f_{\alpha \to o}$

PVS

Die Logik von PVS basiert auf der gerade beschriebenen Typtheorie, beinhaltet aber einige Erweiterungen, die die Mächtigkeit der Sprache wesentlich erhöhen:

- Mehrere Grunddatentypen
- Tupel, Strukturen (Records)
- Prädikaten-Subtypen
- Strukturelle Subtypen
- Abhängige Typen (dependent Types)
- Syntaktische Erweiterungen für algebraische Datentypen

Andere Features erlauben die einfachere Strukturierung von Spezifikationen, z.B. hierarchische, parametrisierbare Theorien.

Typen in PVS

- Basistypen: `number`, `boolean`, ...
- Aufzählungstypen: `{red, green, blue}`
- Funktionstypen: `[number -> number]`
- Record-Typen: `[# flag: boolean, value: number #]`
- Tupel-Typen: `[boolean, number]`
- Cotupel-Typen (disjunkte Summen): `[boolean + number]`

Typen in PVS

- Algebraische Datentypen (und Codatentypen)

```
list[T: TYPE]: DATATYPE BEGIN
      null: null?
      cons(car: T, cdr: list): cons?
END DATATYPE
```

- Prädikaten-Subtypen:
  - `{x: real | x /= 0}`
  - `{f: [real -> real] | injective?(f)}`
  - `{x: T | P(x)}` kann als `(P)` geschrieben werden
- Strukturelle Subtypen:
  - `[# x,y: real, c: color #]` ist Subtyp von `[# x,y: real #]`
  - Updates respektieren Subtypen
- Abhängige Typen (dependent Types) für Funktionen, Tupel, Records und algebraische Datentypen:

```
[n: nat -> {m: nat | m <= n}]
```

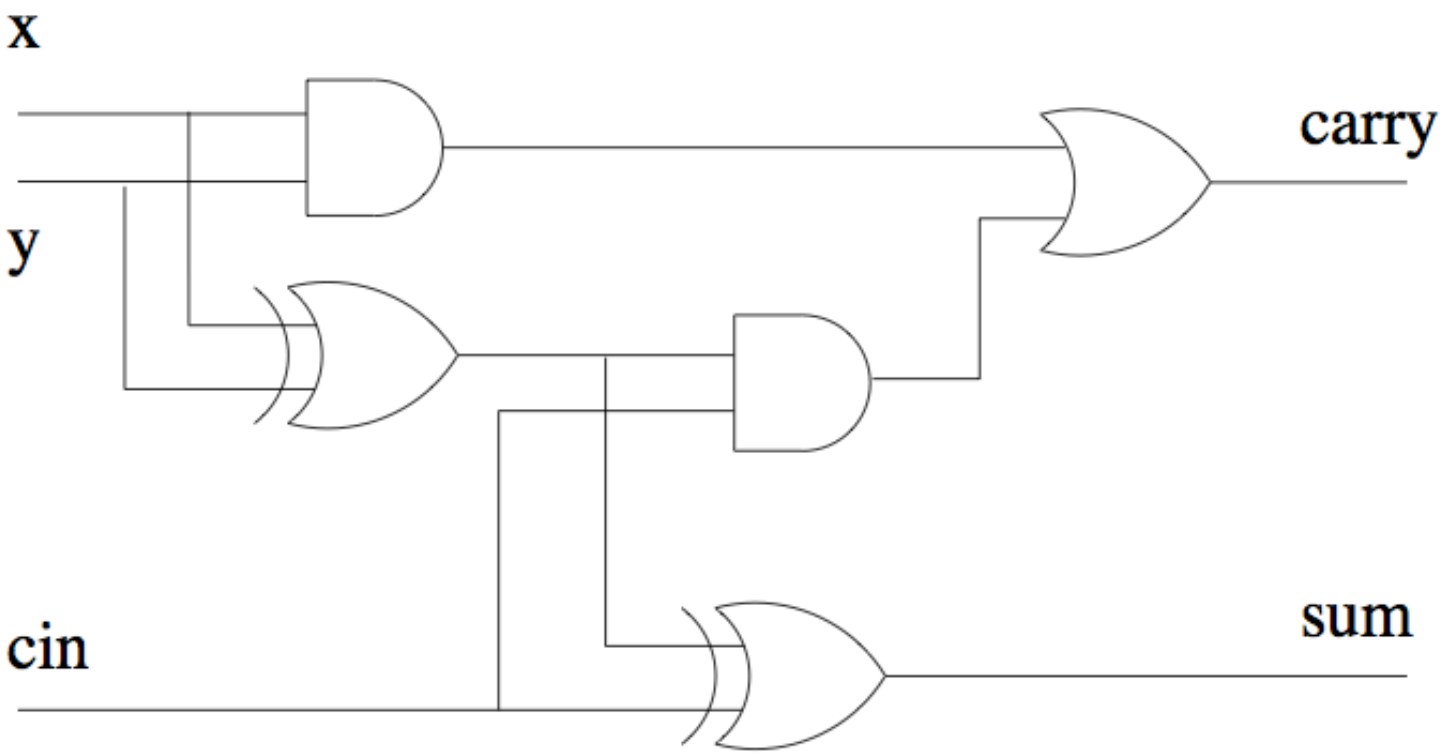
PVS Ausdrücke

- Logik: `TRUE`, `FALSE`, `AND`, `OR`, `XOR`, `NOT`, `IMPLIES`, `FORALL`, `EXISTS`, `=`, ...
- Arithmetik: `+`, `-`, `*`, `/`, `<`, `<=`, `>`, `>=`, `0`, `1`, `2`, ...
- Funktionen:
  - Applikation (`f(x)`)
  - Abstraktion (`LAMBDA (x): A = ...`)
  - Update (`f WITH [(X) := 1]`)
- Hilbert-Operator: `the! (x: nat) p(x)`
- Typumwandlungen

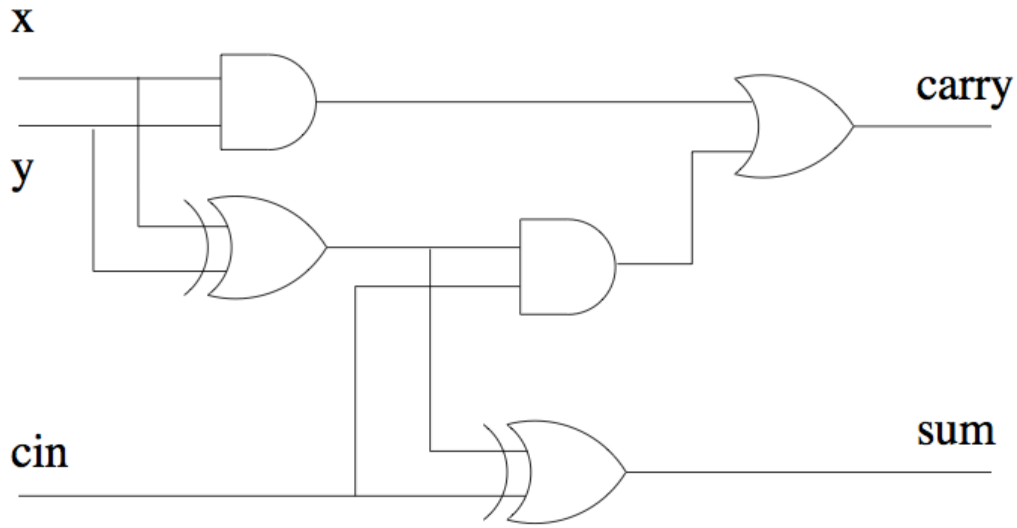
PVS Ausdrücke

- Records: Konstruktion (`(# size := 0 #)`), Selektion (`size(r)`), Update (`r WITH [size := 1]`)
- Tupel: Konstruktion (`(0, 1)`), Selektion (`proj_1(t)` oder `t`1`), Update (`t WITH [1 := 1]`)
- Konditionale: `IF-THEN-ELSE`, `COND`
- Destrukturierung von Records und Tupeln: (`LET ... = ... IN ...`)
- Pattern-Matching von (Co-)Datentypen `CASES`
- Tabellen

Beispiel: Addierer



Beispiel: Addierer




```
FullAdder: THEORY
 BEGIN
  x, y, cin: VAR bool

  FA(x, y, cin): [bool, bool] =
      ((x AND y) OR ((x XOR y) AND cin),
       (x XOR y) XOR cin)

  bool2nat(x): nat = IF x THEN 1 ELSE 0 ENDIF

  FA_corr: THEOREM
    LET (carry, sum) = FA(x, y, cin) IN
      bool2nat(sum) + 2 * bool2nat(carry) =
       bool2nat(x) + bool2nat(y) + bool2nat(cin)
 END FullAdder
```

Beispiel: Addierer

```
FA_corr :

  |-------
{1}   FORALL (cin, x, y: bool):
        LET (carry, sum) = FA(x, y, cin) IN
          bool2nat(sum) + 2 * bool2nat(carry) =
           bool2nat(x) + bool2nat(y) + bool2nat(cin)

Rule? (skolem!)
Skolemizing,
this simplifies to:
FA_corr :

  |-------
{1}   LET (carry, sum) = FA(x!1, y!1, cin!1) IN
        bool2nat(sum) + 2 * bool2nat(carry) =
         bool2nat(x!1) + bool2nat(y!1) + bool2nat(cin!1)
```

Beispiel: Addierer

```
FA_corr :

  |-------
{1}   LET (carry, sum) = FA(x!1, y!1, cin!1) IN
        bool2nat(sum) + 2 * bool2nat(carry) =
         bool2nat(x!1) + bool2nat(y!1) + bool2nat(cin!1)

Rule? (beta)
Applying beta-reduction,
this simplifies to:
FA_corr :

  |-------
{1}  bool2nat(FA(x!1, y!1, cin!1)`2) + 2 * bool2nat(FA(x!1, y!1, cin!1)`1)
       = bool2nat(x!1) + bool2nat(y!1) + bool2nat(cin!1)
```

Beispiel: Addierer

```
FA_corr :

  |-------
{1}  bool2nat(FA(x!1, y!1, cin!1)`2) + 2 * bool2nat(FA(x!1, y!1, cin!1)`1)
       = bool2nat(x!1) + bool2nat(y!1) + bool2nat(cin!1)

Rule? (expand "bool2nat")
Expanding the definition of bool2nat,
this simplifies to:
FA_corr :

  |-------
{1}   IF FA(x!1, y!1, cin!1)`2 THEN 1 ELSE 0 ENDIF +
       2 * IF FA(x!1, y!1, cin!1)`1 THEN 1 ELSE 0 ENDIF
       =
       IF x!1 THEN 1 ELSE 0 ENDIF + IF y!1 THEN 1 ELSE 0 ENDIF +
        IF cin!1 THEN 1 ELSE 0 ENDIF
```

Beispiel: Addierer

```
FA_corr :
  |-------
{1}   IF FA(x!1, y!1, cin!1)`2 THEN 1 ELSE 0 ENDIF +
       2 * IF FA(x!1, y!1, cin!1)`1 THEN 1 ELSE 0 ENDIF
       =
       IF x!1 THEN 1 ELSE 0 ENDIF + IF y!1 THEN 1 ELSE 0 ENDIF +
        IF cin!1 THEN 1 ELSE 0 ENDIF
Rule? (expand "FA")
Expanding the definition of FA,
this simplifies to:
FA_corr :
  |-------
{1}   IF (x!1 XOR y!1) XOR cin!1 THEN 1 ELSE 0 ENDIF +
       2 *
        IF (x!1 AND y!1) OR ((x!1 XOR y!1) AND cin!1) THEN 1 ELSE 0 ENDIF
       =
       IF x!1 THEN 1 ELSE 0 ENDIF + IF y!1 THEN 1 ELSE 0 ENDIF +
        IF cin!1 THEN 1 ELSE 0 ENDIF
```

Beispiel: Addierer

```
FA_corr :
  |-------
{1}   IF (x!1 XOR y!1) XOR cin!1 THEN 1 ELSE 0 ENDIF +
       2 *
        IF (x!1 AND y!1) OR ((x!1 XOR y!1) AND cin!1) THEN 1 ELSE 0 ENDIF
       =
       IF x!1 THEN 1 ELSE 0 ENDIF + IF y!1 THEN 1 ELSE 0 ENDIF +
        IF cin!1 THEN 1 ELSE 0 ENDIF
Rule? (lift-if)
Lifting IF-conditions to the top level,
this simplifies to:
FA_corr :
  |-------
{1}   IF (x!1 XOR y!1) XOR cin!1
        THEN 1 +
             2 *
              IF (x!1 AND y!1) OR ((x!1 XOR y!1) AND cin!1) THEN 1
              ELSE 0
              ENDIF
             =
             IF x!1 THEN 1 ELSE 0 ENDIF + IF y!1 THEN 1 ELSE 0 ENDIF +
```

Beispiel: Addierer

```
...
FA_corr :
  |-------
{1}   IF (x!1 XOR y!1) XOR cin!1
        THEN 1 +
              2 *
               IF (x!1 AND y!1) OR ((x!1 XOR y!1) AND cin!1) THEN 1
               ELSE 0
               ENDIF
              =
              IF x!1 THEN 1 ELSE 0 ENDIF + IF y!1 THEN 1 ELSE 0 ENDIF +
               IF cin!1 THEN 1 ELSE 0 ENDIF
      ELSE 0 +
            2 *
             IF (x!1 AND y!1) OR ((x!1 XOR y!1) AND cin!1) THEN 1
             ELSE 0
             ENDIF
            =
            IF x!1 THEN 1 ELSE 0 ENDIF + IF y!1 THEN 1 ELSE 0 ENDIF +
             IF cin!1 THEN 1 ELSE 0 ENDIF
      ENDIF
```

Beispiel: Addierer

```
FA_corr :
  |-------
{1}   IF (x!1 XOR y!1) XOR cin!1
        ...
      ENDIF

Rule? (prop)
Applying propositional simplification,
this yields  2 subgoals:
FA_corr.1 :

{-1}  (x!1 XOR y!1) XOR cin!1
  |-------
{1}   1 +
       2 *
        IF (x!1 AND y!1) OR ((x!1 XOR y!1) AND cin!1) THEN 1 ELSE 0 ENDIF
       =
       IF x!1 THEN 1 ELSE 0 ENDIF + IF y!1 THEN 1 ELSE 0 ENDIF +
        IF cin!1 THEN 1 ELSE 0 ENDIF
```

Beispiel: Addierer

```
FA_corr.1 :

{-1}  (x!1 XOR y!1) XOR cin!1
  |-------
{1}   1 +
       2 *
        IF (x!1 AND y!1) OR ((x!1 XOR y!1) AND cin!1) THEN 1 ELSE 0 ENDIF
       =
       IF x!1 THEN 1 ELSE 0 ENDIF + IF y!1 THEN 1 ELSE 0 ENDIF +
        IF cin!1 THEN 1 ELSE 0 ENDIF

Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of FA_corr.1.
```

Beispiel: Addierer

```
FA_corr.2 :

  |-------
{1}   (x!1 XOR y!1) XOR cin!1
{2}   0 +
       2 *
        IF (x!1 AND y!1) OR ((x!1 XOR y!1) AND cin!1) THEN 1 ELSE 0 ENDIF
       =
       IF x!1 THEN 1 ELSE 0 ENDIF + IF y!1 THEN 1 ELSE 0 ENDIF +
        IF cin!1 THEN 1 ELSE 0 ENDIF

Rule? (grind)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of FA_corr.2.

Q.E.D.
```

Beispiel: Geordnete Binärbäume

Parametrische Datenstruktur für Binärbäume

```
binary_tree[T : TYPE] : DATATYPE
BEGIN
  leaf : leaf?
  node(val : T, left : binary_tree, right : binary_tree) : noc
END binary_tree
```

Beispiel: Geordnete Binärbäume

Diese Definition generiert eine Typdeklaration, Erkennungsfunktionen (Recognizers), Konstruktoren und Zugriffsfunktionen:

```
binary_tree: TYPE

leaf?: [binary_tree -> boolean]
leaf: (leaf?)

node?: [binary_tree -> boolean]
node: [T, binary_tree, binary_tree -> (node?)]

val: [(node?) -> T]
left: [(node?) -> binary_tree]
right: [(node?) -> binary_tree]
```

Beispiel: Geordnete Binärbäume

Mit DATATYPE deklarierte Typen sind initiale Algebren: für sie gelten Extensionalität („no confusion“) und ein Induktionsschema („no junk“):

```
binary_tree_node_extensionality: AXIOM
  FORALL (node?_var: (node?), node?_var2: (node?)):
    val(node?_var) = val(node?_var2) AND
     left(node?_var) = left(node?_var2) AND
      right(node?_var) = right(node?_var2)
     IMPLIES node?_var = node?_var2;

binary_tree_induction: AXIOM
  FORALL (p: [binary_tree -> boolean]):
    (p(leaf) AND
      (FORALL (node1_var: T, node2_var: binary_tree,
               node3_var: binary_tree):
          p(node2_var) AND p(node3_var) IMPLIES
           p(node(node1_var, node2_var, node3_var))))
     IMPLIES (FORALL (binary_tree_var: binary_tree):
                p(binary_tree_var));
```

Beispiel: Geordnete Binärbäume

Die Funktion `reduce_nat` wird in den folgenden Definitionen zur Definition des Maßes verwendet und wird von PVS aus der `DATATYPE` Deklaration generiert:

```
reduce_nat(leaf?_fun: nat, node?_fun: [[T, nat, nat] -> nat]):
    [binary_tree -> nat] =
  LAMBDA (binary_tree_adtvar: binary_tree):
    CASES binary_tree_adtvar OF
      leaf: leaf?_fun,
      node(node1_var, node2_var, node3_var):
          node?_fun(node1_var,
                    reduce_nat(leaf?_fun, node?_fun)(node2_var),
                    reduce_nat(leaf?_fun, node?_fun)(node3_var))
    ENDCASES;
```

Beispiel: Geordnete Binärbäume

Geordnete Binärbäume können als Theorie definiert werden, die sowohl im Wertetyp als auch in der Ordnungsrelation parametrisch ist:

```
obt [T : TYPE,  <= : (total_order?[T])] : THEORY
BEGIN
IMPORTING binary_tree[T]

A, B, C: VAR binary_tree
x, y, z: VAR T
pp: VAR pred[T]
i, j, k: VAR nat

...
END obt
```

Beispiel: Geordnete Binärbäume

Um ein Maß für die rekursiven Funktionen angeben zu können definieren wir die Funktion `size`. Die Funktion `ordered?` überprüft, ob alle Werte im linken Teilbaum kleiner als der Wert des Knotens, und alle Werte im rechten Teilbaum größer als der Wert des Knotens sind.

```
size(A) : nat =
  reduce_nat(0, (LAMBDA x, i, j: i + j + 1))(A)

ordered?(A) : RECURSIVE bool =
  (IF node?(A) THEN (every((LAMBDA y: y<=val(A)), left(A)) AND
                     every((LAMBDA y: val(A)<=y), right(A)) AND
                     ordered?(left(A)) AND
                     ordered?(right(A)))
        ELSE TRUE ENDIF)
  MEASURE size
```

Beispiel: Geordnete Binärbäume

Einfügen erfolgt wie üblich durch Vergleich mit dem Wert an der Wurzel und Rekursion in den linken oder rechten Unterbaum:

```
insert(x, A): RECURSIVE binary_tree[T] =
  (CASES A OF
    leaf: node(x, leaf, leaf),
    node(y, B, C): (IF x<=y THEN node(y, insert(x, B), C)
                            ELSE node(y, B, insert(x, C))
                    ENDIF)
   ENDCASES)
  MEASURE size(A)
```

Hier sieht man die Verwendung von `CASES` mit Pattern-Matching über den Konstruktoren von `binary_tree`.

Beispiel: Geordnete Binärbäume

Das folgende Lemma beschreibt eine einfache Eigenschaft des Einfügeschritts: wenn `x` das Prädikat `pp` erfüllt und jedes Element aus `A` das Prädikat `pp` erfüllt, dann erfüllt auch jedes Element aus dem durch Einfügen von `x` in `A` entstandenen Baum `pp`:

```
ordered?_insert_step: LEMMA
   pp(x) AND every(pp, A) IMPLIES every(pp, insert(x, A))
```

Beispiel: Geordnete Binärbäume

Das folgende Theorem zeigt die Korrektheit des `insert`-Algorithmus:

```
ordered?_insert: THEOREM
   ordered?(A) IMPLIES ordered?(insert(x, A))
```

Beweis:

```
(""
 (induct-and-simplify "A" :rewrites "ordered?_insert_step")
 (rewrite "ordered?_insert_step")
 (typepred "<=")
 (grind :if-match all))
```

Beispiel: Geordnete Binärbäume

Binärsuche lässt sich ebenfalls leicht implementieren:

```
search(x, A): RECURSIVE bool =
  (CASES A OF
    leaf: FALSE,
    node(y, B, C): (IF x = y THEN TRUE
                    ELSIF x<=y THEN search(x, B)
                      ELSE search(x, C)
                    ENDIF)
   ENDCASES)
  MEASURE size(A)
```

Beispiel: Geordnete Binärbäume

Das folgende Theorem besagt, dass `search` und `insert` auf die gewünschte Weise interagieren:

```
search_insert: THEOREM
  search(y, insert(x, A)) = (x = y OR search(y, A))
```