# Formale Methoden der Softwaretechnik

Christoph Lüth

http://www.informatik.uni-bremen.de/~cxl/



WS 05/06

# Vorlesung vom 24.10.05: Einführung

# Organisatorisches

- Veranstalter: Christoph Lüth

  cxl@informatik.uni-bremen.de

  http://www.informatik.uni-bremen.de/~cxl

  MZH 8050, Tel. 7585

- Termine: Vorlesung: Montag 15– 17 MZH 7260

  Übung: Donnerstag 9 s.t. – 10 8090
  Donnerstag 10 – 11 MZH 8090

# Therac-25

- Neuartiger Linearbeschleuniger in der Strahlentherapie.
  - Computergesteuert (PDP-11, Assembler)
- Fünf Unfälle mit Todesfolge (1985– 1987)
  - Zu hohe Strahlendosis (4000 – 20000 rad, letal 1000 rad)
- Problem: Softwarefehler
  - Ein einzelner Programmierer (fünf Jahre)
  - Alles in Assembler, kein Betriebssystem
  - Programmierer auch Tester (Qualitätskontrolle)

# Ariane-5

# Die Vasa

# Lernziele

1. Modellierung — Formulierung von Spezifikationen

# Lernziele

1. Modellierung — Formulierung von Spezifikationen
2. Verifikation — Beweis von Korrektheit

# Lernziele

1. Modellierung — Formulierung von Spezifikationen
2. Verifikation — Beweis von Korrektheit
3. Formale Entwicklung und formaler Beweis

# Lernziele

1. Modellierung — Formulierung von Spezifikationen
2. Verifikation — Beweis von Korrektheit
3. Formale Entwicklung und formaler Beweis

Darüber hinaus:

- Vertrautheit mit aktuellen Techniken (Isabelle, Z, CASL)

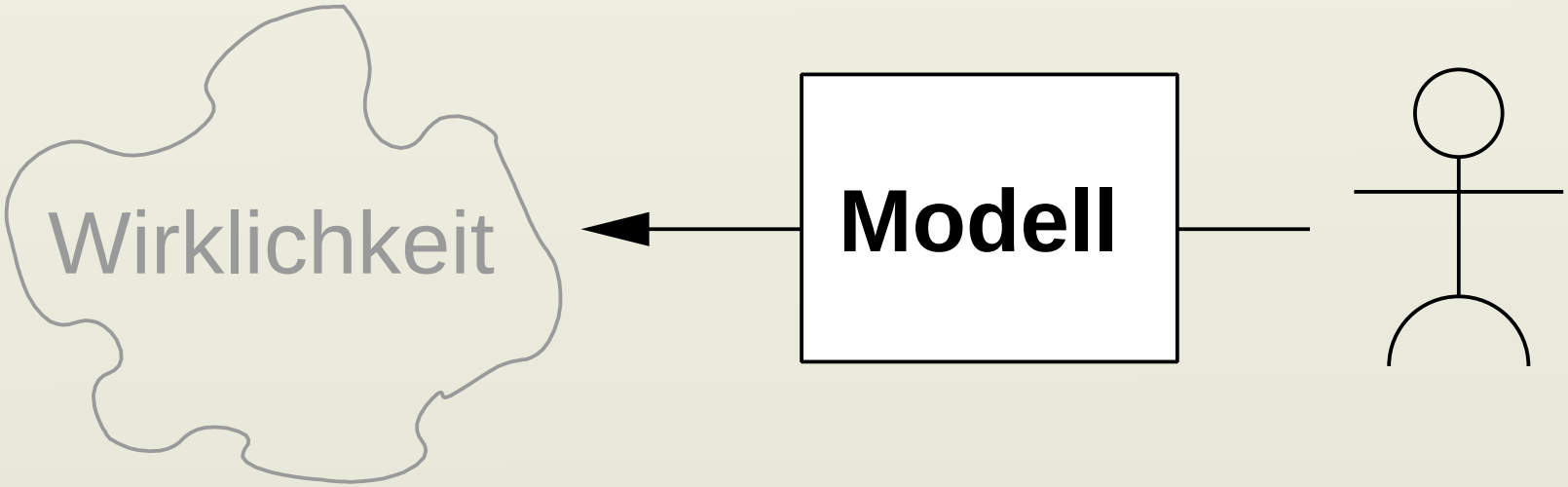
# Themen

- Grundlagen:
  - Formale Logik, formales Beweisen
- Formale Spezifikation
  - Modellbasiert: Z
  - Axiomatisch: CASL
- Verifikation vs. formale Entwicklung
  - Nicht notwendigerweise ein Gegensatz (Djikstra, Gries)

# Plan

- Nächste fünf Wochen:
  - Formale Logik, funktionale Programmierung (Isabelle)
- Bis Weihnachten:
  - Spezifikation imperativer Programme (Z)
- Nach Weihnachten:
  - Grundlagen der Verifikation: weakest precondition
  - Entwicklung durch Verfeinerung und Transformation
  - Software Modelchecking (wenn Zeit)

# Das Problem



# Formale Logik

- Formale (symbolische) Logik: Rechnen mit Symbolen
- Programme: Symbolmanipulation
- Auswertung: Beweis
- Curry-Howard-Isomorphie: funktionale Programme $\cong$ konstruktiver Beweis

# Geschichte

- Gottlob Frege (1848– 1942)
  - ‘Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens’ (1879)
- Georg Cantor (1845– 1918), Bertrand Russel (1872– 1970), Ernst Zermelo (1871– 1953)
  - Einfache Mengenlehre: inkonsistent (Russel’s Paradox)
  - Axiomatische Mengenlehre: Zermelo-Fränkel
- David Hilbert (1862– 1943)
  - Hilbert’s Programm: ‘mechanisierte’ Beweistheorie
- Kurt Gödel (1906– 1978)
  - Vollständigkeitssatz, Unvollständigkeitssätze

# Grundbegriffe der formalen Logik

- Ableitbarkeit $\mathcal{T}h \vdash P$
  - Syntaktische Folgerungung
- Gültigkeit $\mathcal{T}h \models P$
  - Semantische Folgerung — hier nicht relevant
- Klassische Logik: $P \vee \neg P$
- Entscheidbarkeit
  - Aussagenlogik
- Konsistenz: $\mathcal{T}h \nvdash \bot$
  - Nicht alles ableitbar
- Vollständigkeit: jede gültige Aussage ableitbar
  - Prädikatenlogik erster Stufe

# Unvollständigkeit

- Gödels 1. Unvollständigkeitssatz:
  - Jede Logik, die Peano-Arithmetik formalisiert, ist entweder inkonsistent oder unvollständig.

# Unvollständigkeit

- Gödels 1. Unvollständigkeitssatz:
  - Jede Logik, die Peano-Arithmetik formalisiert, ist entweder inkonsistent oder unvollständig.
- Gödels 2. Unvollständigkeitssatz:
  - Jeder Logik, die ihre eigene Konsistenz beweist, ist inkonsistent.

# Unvollständigkeit

- Gödels 1. Unvollständigkeitssatz:
  - Jede Logik, die Peano-Arithmetik formalisiert, ist entweder inkonsistent oder unvollständig.
- Gödels 2. Unvollständigkeitssatz:
  - Jeder Logik, die ihre eigene Konsistenz beweist, ist inkonsistent.
- Auswirkungen:
  - Hilbert's Programm terminiert nicht.
  - Programme nicht vollständig spezifierbar.
  - Spezifikationssprachen immer unvollständig (oder uninteressant).

# Unvollständigkeit

- Gödels 1. Unvollständigkeitssatz:
  - Jede Logik, die Peano-Arithmetik formalisiert, ist entweder inkonsistent oder unvollständig.
- Gödels 2. Unvollständigkeitssatz:
  - Jeder Logik, die ihre eigene Konsistenz beweist, ist inkonsistent.
- Auswirkungen:
  - Hilbert's Programm terminiert nicht.
  - Programme nicht vollständig spezifierbar.
  - Spezifikationssprachen immer unvollständig (oder uninteressant).
  - Mit anderen Worten: Es bleibt spannend.

# Vorlesung vom 31.10.05: Formale Logik und natürliches Schließen

# Fahrplan

- Einführung in die formale Logik
- Aussagenlogik
  - Beispiel für eine einfache Logik
  - Guter Ausgangspunkt
- Natürliches Schließen
  - Wird auch von Isabelle verwendet.
- Buchempfehlung:

  Dirk van Dalen: Logic and Structure. Springer Verlag, 2004.

# Formale Logik

- Ziel: Formalisierung von Folgerungen wie

  Wenn es regnet, wird die Straße nass.

# Formale Logik

- Ziel: Formalisierung von Folgerungen wie

  Wenn es regnet, wird die Straße nass.
  Es regnet.

# Formale Logik

- Ziel: Formalisierung von Folgerungen wie

  Wenn es regnet, wird die Straße nass. Nachts ist es dunkel.
  Es regnet.
  Also ist die Straße nass.

# Formale Logik

- Ziel: Formalisierung von Folgerungen wie

  Wenn es regnet, wird die Straße nass.
  Es regnet.
  Also ist die Straße nass.

  Nachts ist es dunkel.
  Es ist hell.

# Formale Logik

- Ziel: Formalisierung von Folgerungen wie

  Wenn es regnet, wird die Straße nass.
  Es regnet.
  Also ist die Straße nass.

  Nachts ist es dunkel.
  Es ist hell.
  Also ist es nicht nachts.

- Eine Logik besteht aus
  - Einer Sprache $\mathcal{L}$ von Formeln (Aussagen)
  - Schlußregeln (Folgerungsregeln) auf diesen Formeln.

  Damit: Gültige ("wahre") Aussagen berechnen.

# Beispiel für eine Logik

- Sprache $\mathcal{L} = \{\clubsuit, \spadesuit, \heartsuit, \diamondsuit\}$

# Beispiel für eine Logik

- Sprache $\mathcal{L} = \{\clubsuit, \spadesuit, \heartsuit, \diamondsuit\}$
- Schlußregeln:

$$\frac{\diamondsuit}{\clubsuit}\,\alpha \qquad \frac{\diamondsuit}{\spadesuit}\,\beta \qquad \frac{\clubsuit \quad \spadesuit}{\heartsuit}\,\gamma \qquad \frac{\begin{array}{c}[\diamondsuit] \\ \vdots \\ \heartsuit\end{array}}{\heartsuit}\,\delta$$

- Beispielableitung: $\heartsuit$

# Aussagenlogik

Sprache $\mathcal{P}rop$ gegeben durch:

- Variablen $V \subseteq \mathcal{P}rop$ (Menge $V$ gegeben)
- $false \in \mathcal{P}rop$
- Wenn $\phi, \psi \in \mathcal{P}rop$, dann $\phi \wedge \psi \in \mathcal{P}rop$, $\phi \vee \psi \in \mathcal{P}rop$, $\phi \longrightarrow \psi \in \mathcal{P}rop$, $\phi \longleftrightarrow \psi \in \mathcal{P}rop$
- Wenn $\phi \in \mathcal{P}rop$, dann $\neg\phi \in \mathcal{P}rop$.

# Wann ist eine Formel gültig?

- Semantische Gültigkeit $\models P$ : Wahrheitstabellen etc.
  - Wird hier nicht weiter verfolgt.
- Syntaktische Gültigkeit $\vdash P$: formale Ableitung,
  - Natürliches Schließen
  - Sequenzenkalkül
  - Andere (Hilbert-Kalkül, gleichungsbasierte Kalküle, etc.)

# Natürliches Schließen

- Vorgehensweise: Erst Kalkül nur für $\wedge, \longrightarrow, \mathit{false}$, dann Erweiterung auf alle Konnektive.
- Für jedes Konnektiv: Einführungs- und Eliminitationsregel
- NB: konstruktiver Inhalt der meisten Regeln

# Natürliches Schließen — Die Regeln

$$\frac{\phi \quad \psi}{\phi \wedge \psi} \wedge I \qquad \frac{\phi \wedge \psi}{\phi} \wedge E_1 \qquad \frac{\phi \wedge \psi}{\psi} \wedge E_2$$

$$\frac{\begin{array}{c}[\phi] \\ \vdots \\ \psi\end{array}}{\phi \longrightarrow \psi} \longrightarrow I \qquad \frac{\phi \quad \phi \longrightarrow \psi}{\psi} \longrightarrow E$$

$$\frac{false}{\phi}\, false \qquad \frac{\begin{array}{c}[\phi \longrightarrow false] \\ \vdots \\ false\end{array}}{\phi}\, \mathsf{raa}$$

# Konsistenz

Def: $\Gamma$ konsistent gdw. $\Gamma \nvdash \mathit{false}$

Lemma: Folgende Aussagen sind äquivalent:

(i) $\Gamma$ konsistent

(ii) Es gibt kein $\phi$ so dass $\Gamma \vdash \phi$ und $\Gamma \vdash \neg\phi$

(iii) Es gibt ein $\phi$ so dass $\Gamma \nvdash \phi$

Satz: Aussagenlogik mit natürlichem Schließen ist konsistent.

# Die fehlenden Konnektive

- Einführung als Abkürzung:

$$\neg\phi \stackrel{def}{=} \phi \longrightarrow false$$

$$\phi \vee \psi \stackrel{def}{=} \neg(\neg\phi \wedge \neg\psi)$$

$$\phi \longleftrightarrow \psi \stackrel{def}{=} (\phi \longrightarrow \psi) \wedge (\psi \longrightarrow \phi)$$

- Ableitungsregeln als Theoreme.

# Die fehlenden Schlußregeln

$$\frac{\phi}{\phi \vee \psi} \vee I_1 \qquad \frac{\psi}{\phi \vee \psi} \vee I_2$$

$$\frac{\phi \vee \psi \qquad \begin{array}{c}[\phi] \\ \vdots \\ \sigma\end{array} \qquad \begin{array}{c}[\psi] \\ \vdots \\ \sigma\end{array}}{\sigma} \vee E$$

$$\frac{\begin{array}{c}[\phi] \\ \vdots \\ false\end{array}}{\neg\phi} \neg I \qquad \frac{\phi \quad \neg\phi}{false} \neg E$$

$$\frac{\phi \longrightarrow \psi \quad \psi \longrightarrow \phi}{\phi \longleftrightarrow \psi} \longleftrightarrow I \qquad \frac{\phi \quad \phi \longleftrightarrow \psi}{\psi} \longleftrightarrow E_1 \qquad \frac{\psi \quad \phi \longleftrightarrow \psi}{\phi} \longleftrightarrow E_2$$

# Zusammenfassung

- Formale Logik formalisiert das (natürlichsprachliche) Schlußfolgern
- Logik: Aussagen plus Schlußregeln (Kalkül)
- Aussagenlogik: Aussagen mit $\wedge$, $\longrightarrow$, $false$
  - $\neg$, $\vee$, $\longleftrightarrow$ als abgeleitete Operatoren
- Natürliches Schließen: intuitiver Kalkül
- Aussagenlogik konsistent, vollständig, entscheidbar.
- Nächstes Mal: Quantoren, HOL.

# Vorlesung vom 07.11.05: Prädikatenlogik erster Stufe (Vom Umgang mit Quantoren)

# Fahrplan

- Logik mit Quantoren
- Von Aussagenlogik zur Prädikatenlogik
- Natürliches Schließen mit Quantoren
- Die Notwendigkeit von Logik höherer Stufe

# Wo sind wir?

- Aussagenlogik
- Prädikatenlogik
- Logik höherer Stufe
- Isabelle/HOL

# Prädikatenlogik

- Beschränkung der Aussagenlogik:

  Alle Quadratzahlen sind positiv,

# Prädikatenlogik

- Beschränkung der Aussagenlogik:

  Alle Quadratzahlen sind positiv, 9 ist eine Quadratzahl,

# Prädikatenlogik

- Beschränkung der Aussagenlogik:

  Alle Quadratzahlen sind positiv, 9 ist eine Quadratzahl,
  also ist 9 positiv.

  Nicht in Aussagenlogk formalisierbar.

# Prädikatenlogik

- Beschränkung der Aussagenlogik:

  Alle Quadratzahlen sind positiv, 9 ist eine Quadratzahl,
  also ist 9 positiv.

  Nicht in Aussagenlogk formalisierbar.

- Ziel: Formalisierung von Aussagen wie

  Alle Zahlen sind ein Produkt von Primfaktoren.

  Es gibt keine größte Primzahl.

# Erweiterung der Sprache

- Terme beschreiben die zu formalisierenden Objekte.
- Formeln sind logische Aussagen.
- Unser Alphabet:
  - Prädikatensymbole: $P_1, \ldots, P_n, \bullet =$ mit Arität $ar(P_i) \in \mathbb{N}$, $ar(\bullet =) = 2$
  - Funktionssymbole: $f_1, \ldots, f_m$ mit Arität $ar(t_i) \in \mathbb{N}$
  - Menge $X$ von Variablen (abzählbar viele)
  - Konnektive: $\wedge, \longrightarrow, \mathit{false}, \forall$, abgeleitet: $\vee, \longleftrightarrow, \neg, \longleftrightarrow, \exists$

# Terme

Menge $\mathcal{T}erm$ der Terme gegeben durch:

- Variablen: $X \subseteq \mathcal{T}erm$
- Funktionssymbol $f$ mit $ar(f) = n$ und $t_1, \ldots, t_n \in \mathcal{T}erm$, dann $f(t_1, \ldots, t_n) \in \mathcal{T}erm$
- Sonderfall: $n = 0$, dann ist $f$ eine Konstante, $f \in \mathcal{T}erm$

# Formeln

Menge $\mathcal{F}orm$ der Formeln gegeben durch:

- $false \in \mathcal{F}orm$
- Wenn $\phi \in \mathcal{F}orm$, dann $\neg\phi \in \mathcal{F}orm$
- Wenn $\phi, \psi \in \mathcal{F}orm$, dann $\phi \wedge \psi \in \mathcal{F}orm$, $\phi \vee \psi \in \mathcal{F}orm$, $\phi \longrightarrow \psi \in \mathcal{F}orm$, $\phi \longleftrightarrow \psi \in \mathcal{F}orm$

# Formeln

Menge $\mathcal{F}orm$ der Formeln gegeben durch:

- $false \in \mathcal{F}orm$
- Wenn $\phi \in \mathcal{F}orm$, dann $\neg\phi \in \mathcal{F}orm$
- Wenn $\phi, \psi \in \mathcal{F}orm$, dann $\phi \wedge \psi \in \mathcal{F}orm$, $\phi \vee \psi \in \mathcal{F}orm$, $\phi \longrightarrow \psi \in \mathcal{F}orm$, $\phi \longleftrightarrow \psi \in \mathcal{F}orm$
- Wenn $\phi \in \mathcal{F}orm, x \in X$, dann $\forall\, x.\phi \in \mathcal{F}orm, \exists\, x.\phi \in \mathcal{F}orm$
- Prädikatensymbol $p$ mit $ar(p) = m$ und $t_1, \ldots, t_m \in \mathcal{T}erm$, dann $p(t_1, \ldots, t_m) \in \mathcal{F}orm$
  - Sonderfall: $t_1, t_2 \in \mathcal{T}erm$, dann $t_1 \bullet = t_2 \in \mathcal{F}orm$

# Beispielaussagen

- Alle Zahlen sind gerade oder ungerade.

# Beispielaussagen

- Alle Zahlen sind gerade oder ungerade.
- Keine Zahl ist gerade und ungerade.

# Beispielaussagen

- Alle Zahlen sind gerade oder ungerade.
- Keine Zahl ist gerade und ungerade.
- Es gibt keine größte Primzahl.

# Beispielaussagen

- Alle Zahlen sind gerade oder ungerade.
- Keine Zahl ist gerade und ungerade.
- Es gibt keine größte Primzahl.
- Für jede Primzahl gibt es eine, die größer ist.

# Beispielaussagen

- Alle Zahlen sind gerade oder ungerade.
- Keine Zahl ist gerade und ungerade.
- Es gibt keine größte Primzahl.
- Für jede Primzahl gibt es eine, die größer ist.
- Eine Funktion $f$ ist stetig an der Stelle $x_0$, gdw. es für jedes $\varepsilon > 0$ ein $\delta > 0$ gibt, so dass für alle $x$ mit $\mid x - x_0 \mid < \delta$ gilt $\mid f(x) - f(x_0) \mid < \epsilon$.

# Freie und gebundene Variable

- Variablen in $t \in \mathcal{T}erm, p \in \mathcal{F}orm$ sind frei, gebunden, oder bindend.
  - $x$ bindend in $\forall\, x.\phi$, $\exists\, x.\psi$
  - Für $\forall\, x.\phi, \exists\, \phi x$ ist $x$ in Teilformel $\phi$ gebunden
  - Ansonsten ist $x$ frei
- $FV(\phi)$: Menge der freien Variablen in $\phi$
- Beispiel:

$$(q(x) \vee \exists\, x.\forall\, y.p(f(x), z) \wedge q(a)) \vee \forall\, r(x, z, g(x))$$

# Natürliches Schließen mit Quantoren

$$\frac{\phi}{\forall\, x.\phi}\,\forall\, I \quad (*) \qquad\qquad \frac{\forall\, x.\phi}{\phi\left[\begin{smallmatrix} t \\ x \end{smallmatrix}\right]}\,\forall\, E \quad (\dagger)$$

(*) Eigenvariablenbedingung:
$x$ nicht frei in offenen Vorbedingungen von $\phi$

($\dagger$) Ggf. Umbenennung in $T$ Substitution

- Gegenbeispiele für verletzte Seitenbedingungen

# Ersetzung

- $t\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right]$ ist Ersetzung von $x$ durch $s$ in $t$
- Definiert durch strukturelle Induktion:

$$y\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right] \stackrel{def}{=} \begin{cases} s & x = y \\ y & x \neq y \end{cases}$$

$$f(t_1, \ldots, t_n)\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right] \stackrel{def}{=} f(t_1\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right], \ldots, t_n\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right])$$

$$false\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right] \stackrel{def}{=} false \qquad (\phi \wedge \psi)\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right] \stackrel{def}{=} \phi\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right] \wedge \psi\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right] \qquad \ldots$$

$$p(t_1, \ldots, t_n)\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right] \stackrel{def}{=} p(t_1\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right], \ldots, t_n\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right])$$

$$(\forall\, y.\phi)\left[\begin{smallmatrix} s \\ x \end{smallmatrix}\right] \stackrel{def}{=} \begin{cases} \forall\, y.\phi & x = y \\ \forall\, y.(\phi\left[\begin{smallmatrix} s' \\ x \end{smallmatrix}\right]) & x \neq y, \text{ mit } s' \stackrel{def}{=} s\left[\begin{smallmatrix} y' \\ y \end{smallmatrix}\right], \\ & y' \text{ frische Variable} \end{cases}$$

# Der Existenzquantor

$$\exists\, x.\phi \stackrel{def}{=} \neg \forall\, x.\neg\phi$$

$$\frac{\phi\left[\begin{smallmatrix} t \\ x \end{smallmatrix}\right]}{\exists\, x.\phi}\,\exists\, I \quad (\dagger) \qquad\qquad \frac{\exists\, x.\phi \qquad \begin{array}{c}[\phi] \\ \vdots \\ \psi\end{array}}{\psi}\,\exists\, E \quad (*)$$

(*) Eigenvariablenbedingung:

$x$ nicht frei in $\psi$, oder einer offenenen Vorbedingung außer $\phi$

(†) Ggf. Umbenennung durch Substitution

# Regeln für die Gleichheit

- Reflexivität, Symmetrie, Transitivität:

$$\frac{}{x = x}\,\text{refl} \qquad \frac{x = y}{y = x}\,\text{sym} \qquad \frac{x = y \quad y = z}{x = z}\,\text{trans}$$

- Kongruenz:

$$\frac{x_1 = y_1, \ldots, x_n = y_n}{f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)}\,\text{conf}$$

- Substitutivität:

$$\frac{x_1 = y_1, \ldots, x_m = y_m \quad P(x_1, \ldots, x_n)}{P(y_1, \ldots, y_m)}\,\text{subst}$$

# Zusammenfassung

- Prädikatenlogik: Erweiterung der Aussagenlogik um
  - Konstanten- und Prädikatensymbole
  - Gleichheit
  - Quantoren
- Das natürliche Schließen mit Quantoren
  - Variablenbindungen — Umbenennungen bei Substitution
  - Eigenvariablenbedingung
- Grenzen der Prädikatenlogik erster Stufe:
  - z.B. Peano-Axiome
- Deshalb das nächste Mal: Logik höherer Stufe
  - Die ganze Mathematik in sieben Axiomen.

# Vorlesung vom 14.11.05: Logik höherer Stufe I

# Fahrplan

- Syntaktische Basis: Der $\lambda$-Kalkül
  - Typen und Terme
  - Substitution
  - Reduktion

# Wo sind wir?

- Aussagenlogik
- Prädikatenlogik
- Logik höherer Stufe
- Isabelle/HOL

# Der $\lambda$-Kalkül

- In den 30'ern von Alonzo Church als theoretisches Berechhnungsmodell erfunden.
- In den 60'ern Basis von Lisp (John McCarthy)
- Mathematische Basis von funktionalen Sprachen (Haskell etc.)
- Hier: Grundlage der Syntax ("higher-order abstract syntax")
  - Typisierung
  - Gebundene Variablen

# Der einfach getypte $\lambda$-Kalkül ($\lambda^{\rightarrow}$)

Typen $\mathcal{T}ype$ gegeben durch

- Typkonstanten: $c \in \mathcal{C}_{\mathcal{T}ype}$ (Menge $\mathcal{C}_{\mathcal{T}ype}$ gegeben)
- Funktionen: $s, t \in \mathcal{T}ype$ dann $s \Rightarrow t$ in $\mathcal{T}ype$

Terme $\mathcal{T}erm$ gegeben durch

- Konstanten: $c \in \mathcal{C}$
- Variablen: $v \in \mathcal{V}$
- Applikation: $s, t \in \mathcal{T}erm$ dann $st \in \mathcal{T}erm$
- Abstraktion: $x \in \mathcal{V}, \tau \in \mathcal{T}ypet \in \mathcal{T}erm$ dann $\lambda\, x^{\tau}.t \in \mathcal{T}erm$
  - Typ $\tau$ kann manchmal berechnet werden.
  - Ohne Typen: ungetypter $\lambda$-Kalkül

# Substitution

- $s\left[\begin{smallmatrix} t \\ x \end{smallmatrix}\right]$ ist Ersetzung von $x$ durch $t$ in $s$
- Definiert durch strukturelle Induktion:

$$
\begin{aligned}
c\left[\begin{smallmatrix} t \\ x \end{smallmatrix}\right] &\stackrel{def}{=} c \\
y\left[\begin{smallmatrix} t \\ x \end{smallmatrix}\right] &\stackrel{def}{=} \begin{cases} t & x = y \\ y & x \neq y \end{cases} \\
(rs)\left[\begin{smallmatrix} t \\ x \end{smallmatrix}\right] &\stackrel{def}{=} r\left[\begin{smallmatrix} t \\ x \end{smallmatrix}\right] s\left[\begin{smallmatrix} t \\ x \end{smallmatrix}\right] \\
(\lambda\, z.s)\left[\begin{smallmatrix} t \\ x \end{smallmatrix}\right] &\stackrel{def}{=} \begin{cases} \lambda\, z.s & x = z \\ \lambda\, z.s\left[\begin{smallmatrix} t \\ x \end{smallmatrix}\right] & x \neq z, \quad z \notin FV(t) \text{ or } x \notin FV(s) \\ \lambda\, y.s\left[\begin{smallmatrix} y \\ z \end{smallmatrix}\right]\left[\begin{smallmatrix} t \\ x \end{smallmatrix}\right] & x \neq z, \quad z \in FV(t),\ x \in FV(s), \\ & \qquad\qquad y \notin FV(ts) \end{cases}
\end{aligned}
$$

# $\beta$-Reduktion

- $\beta$-Reduktion: $(\lambda\, x.s)t \longrightarrow_\beta s\left[\begin{smallmatrix} t \\ x \end{smallmatrix}\right]$
- Reduktion im Kontext:

$$\frac{s \longrightarrow_\beta s'}{st \longrightarrow_\beta s't} \qquad \frac{t \longrightarrow_\beta t'}{st \longrightarrow_\beta st'} \qquad \frac{t \longrightarrow_\beta t'}{\lambda\, x.t \longrightarrow_\beta \lambda\, x.t'}$$

- $\longrightarrow_\beta^*$: Transitiv-Reflexiver Abschluss von $\longrightarrow_\beta$
- Exkurs: Der ungetypte Lambda-Kalkül als Berechnungsmodell
  - Reduktion als Berechnung, Fixpunkte, Church-Numerale

# Typisierung

- Term $t$ hat Typ $\tau$ in einem Kontext $\Gamma$ und einer Signatur $\Sigma$:

  $$\Gamma \vdash_\Sigma t : \tau$$

- Kontext: $x_1 : \tau_1, \ldots, x_n : \tau_n \ (x_i \in \mathcal{V})$
  - Typisierung von Variablen
- Signatur: $c_1 : \tau_1, \ldots, c_m : \tau_m \ (c_i \in \mathcal{C})$
  - Typisierung der Konstanten
- Vergleiche Haskell etc.

# Typisierung

- Die Regeln:

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash_\Sigma c : \tau}\, CONST \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash_\Sigma x : \tau}\, VAR$$

$$\frac{\Gamma \vdash_\Sigma s : \sigma \Rightarrow \tau \quad \Gamma \vdash_\Sigma t : \sigma}{\Gamma \vdash_\Sigma st : \tau}\, APP \qquad \frac{\Gamma, x : \sigma \vdash_\Sigma t : \tau}{\Gamma \vdash_\Sigma \lambda\, x^\sigma . t : \sigma \Rightarrow \tau}\, ABST$$

- Lemma: $\beta$-Reduktion von getypten Termen stark normalisierend
- Lemma: $\beta$-Reduktion bewahrt Typ (subject reduction)

# Äquivalenzen

- Äquivalenz: symmetrischer Abschluß der Reduktion
- $\beta$-Äquivalenz:

  $$=_\beta \stackrel{def}{=} (\rightarrow_\beta^* \cup {}_\beta^* \leftarrow)^*$$

  - $s =_\beta t$ iff. $\exists\, u. s \rightarrow_\beta^* u, t \rightarrow_\beta^* u$ (Church-Rosser)
- $\alpha$-Äquivalenz: Name von gebundenen Variablen unerheblich

  $$\lambda\, x.t =_\alpha \lambda\, y.t \begin{bmatrix} y \\ x \end{bmatrix} \quad y \notin FV(t)$$

  - Implizit (deBruijn-Indizes)

- $\eta$-Äquivalenz: “punktfreie” Notation

$$(\lambda\, x.tx) =_\eta t \quad x \notin FV(t)$$

# Aussagenlogik in $\lambda^{\rightarrow}$

- Einbettung der Syntax
- Basistypen:
  $\mathcal{C}_{\mathcal{T}ype} = \{o\}$
- Konstanten:
  $\Sigma_P = \{false : o, and : o \Rightarrow o \Rightarrow o, impl : o \Rightarrow o \Rightarrow o\}$
- $\phi \in \mathcal{P}rop \longleftrightarrow \vdash_{\Sigma_P} t : o$
- Beispiel: $false \longrightarrow (a \wedge a) \in \mathcal{P}rop \longleftrightarrow (impl\ false)((and\ a)\ a)$

# Prädikatenlogik in $\lambda^{\rightarrow}$

- Basistypen:
  $\mathcal{C}_{\mathcal{T}ype} = \{i, o\}$
- Konstanten

$$\Sigma_T = \{0 : i, s : i \Rightarrow i, plus : i \Rightarrow i \Rightarrow i\}$$
$$\Sigma_A = \{\ eq : i \Rightarrow i \Rightarrow o, false : o, and : o \Rightarrow o \Rightarrow o, \ldots,$$
$$all : (i \Rightarrow o) \Rightarrow o, ex : (i \Rightarrow o) \Rightarrow o\}$$
$$\Sigma = \Sigma_T \cup \Sigma_A$$

- $\phi \in \mathcal{F}orm \longleftrightarrow \vdash_\Sigma t : o$
- Beispiel: $\forall\, x.\, \exists\, y.(x = y) \longleftrightarrow all\ (\lambda\, x.ex\ (\lambda\, y.(eq\ x)\ y))$

# Zusammenfassung

- Der $\lambda$-Kalkül:
  - Ein einfaches Berechnungsmodell
  - Meta-Sprache mit Bindungen und Typisierung
  - $\alpha-$, $\beta$, $\eta$-Konversion.
- Einbettung von Syntax (higher-order abstract syntax):
  - Aussagenlogik
  - Prädikatenlogik
- Nächstes Mal: Logik höherer Stufe, eingebettet in $\lambda$-Kalkül.

# Vorlesung vom 21.11.05: Logik höherer Stufe II

# Organisatorisches

- Vorlesung nächste Woche (28.11.05) fällt aus!
- Ersatzvorlesung: diese Woche Do 10-12

# Fahrplan

- Letzte Woche:
  - Der $\lambda$-Kalkül
  - Einbettung von Logiken in den $\lambda$-Kalkül
- Heute: Logik höherer Stufe
  - Typen und Terme
  - Die Basis-Axiome
  - Definierte Operatoren
  - Konservative Erweiterung

# Wo sind wir?

- Aussagenlogik
- Prädikatenlogik
- Logik höherer Stufe
- Isabelle/HOL

# Logik höherer Stufe

- Ziel: Formalisierung von Mathematik
  - "Logik für Erwachsene"
- Problem: Mögliche Inkonsistenz (Russel's Paradox)
- Lösung: Restriktion vs. Ausdrucksstärke
- Alternative Grundlagen:
  - Andere Typtheorien (Martin-Löf, Calculus of Constructions)
  - Ungetypte Mengenlehre (ZFC)
- HOL: guter Kompromiss, weit verbreitet.
  - Klassische Logik höherer Stufe nach Church
  - Schwächer als ZFC, stärker als Typtheorien

# Warum Logik höherer Stufe?

- Logik 1. Stufe: Quantoren über Terme

$$\forall\, x, y.x = y \longrightarrow y = x$$

# Warum Logik höherer Stufe?

- Logik 1. Stufe: Quantoren über Terme

$$\forall\, x, y.x = y \longrightarrow y = x$$

- Logik 2. Stufe: Quantoren über Prädikaten und Funktionen

$$\forall\, P.(P(0) \wedge \forall\, x.P(x) \longrightarrow P(Sx)) \longrightarrow \forall\, x.Px$$

# Warum Logik höherer Stufe?

- Logik 1. Stufe: Quantoren über Terme

$$\forall\, x, y.x = y \longrightarrow y = x$$

- Logik 2. Stufe: Quantoren über Prädikaten und Funktionen

$$\forall\, P.(P(0) \wedge \forall\, x.P(x) \longrightarrow P(Sx)) \longrightarrow \forall\, x.Px$$

- Logik 3. Stufe: Quantoren über Argumenten von Prädikaten

# Warum Logik höherer Stufe?

- Logik 1. Stufe: Quantoren über Terme

$$\forall\, x, y.x = y \longrightarrow y = x$$

- Logik 2. Stufe: Quantoren über Prädikaten und Funktionen

$$\forall\, P.(P(0) \wedge \forall\, x.P(x) \longrightarrow P(Sx)) \longrightarrow \forall\, x.Px$$

- Logik 3. Stufe: Quantoren über Argumenten von Prädikaten
- Logik höherer Stufe: alle endlichen Quantoren
  - Keine wesentlichen Vorteile von Logik 2. Ordnung

# Warum Logik höherer Stufe?

- Logik 1. Stufe: Quantoren über Terme

$$\forall\, x, y.x = y \longrightarrow y = x$$

- Logik 2. Stufe: Quantoren über Prädikaten und Funktionen

$$\forall\, P.(P(0) \wedge \forall\, x.P(x) \longrightarrow P(Sx)) \longrightarrow \forall\, x.Px$$

- Logik 3. Stufe: Quantoren über Argumenten von Prädikaten
- Logik höherer Stufe: alle endlichen Quantoren
  - Keine wesentlichen Vorteile von Logik 2. Ordnung
- Wichtig: Vermeidung von Inkonsistenzen
  - Unterscheidung zwischen Termen und Aussagen

# Typen

Typen $\mathcal{T}ype$ gegeben durch

- Typkonstanten: $c \in \mathcal{C}_{\mathcal{T}ype}$ (Menge $\mathcal{C}_{\mathcal{T}ype}$ gegeben)
  - $Prop, Bool \in \mathcal{C}_{\mathcal{T}ype}$: $Prop$ alle Terme, $Bool$ alle Aussagen
- Typvariablen: $\alpha \in \mathcal{V}_{\mathcal{T}ype}$ (Menge $\mathcal{V}_{\mathcal{T}ype}$ gegeben)
- Funktionen: $s, t \in \mathcal{T}ype$ dann $s \Rightarrow t$ in $\mathcal{T}ype$

Konvention: Funktionsraum nach rechts geklammert

$\alpha \Rightarrow \beta \Rightarrow \gamma$ für $\alpha \Rightarrow (\beta \Rightarrow \gamma)$

# Terme

Terme $\mathcal{T}erm$ gegeben durch

- Konstanten: $c \in \mathcal{C}$
- Variablen: $v \in \mathcal{V}$
- Applikation: $s, t \in \mathcal{T}erm$ dann $st \in \mathcal{T}erm$
- Abstraktion: $x \in \mathcal{V}, t \in \mathcal{T}erm$ dann $\lambda\, x.t \in \mathcal{T}erm$

Konventionen: Applikation links geklammert, mehrfache Abstraktion
$\lambda\, xyz.fxyz$ für $\lambda\, x.\, \lambda\, y.\, \lambda\, z.((fx)y)z$

# HOL: Basis-Syntax

$$
\begin{array}{ll}
\neg & :: Bool \Rightarrow Bool \\
true & :: Bool \\
false & :: Bool \\
if & :: Bool \Rightarrow \alpha \Rightarrow \alpha \Rightarrow \alpha \\
\forall & :: (\alpha \Rightarrow Bool) \Rightarrow Bool \\
\exists & :: (\alpha \Rightarrow Bool) \Rightarrow Bool \\
\iota & :: (\alpha \Rightarrow Bool) \Rightarrow \alpha \\
= & :: \alpha \Rightarrow \alpha \Rightarrow Bool \\
\wedge & :: Bool \Rightarrow Bool \Rightarrow Bool \\
\vee & :: Bool \Rightarrow Bool \Rightarrow Bool \\
\longrightarrow & :: Bool \Rightarrow Bool \Rightarrow Bool
\end{array}
$$

- Einbettung (wird weggelassen)
  $trueprop :: Bool \Rightarrow Ind$
- Basis-Operatoren: $\forall, \longrightarrow, =$
- Syntaktische Konventionen:
  - Bindende Operatoren: $\forall$, $\exists$, $\iota$
    $\forall\, x.P \equiv \forall(\lambda\, x.P)$
  - Infix-Operatoren: $\wedge, \vee, \longrightarrow, =$
  - Mixfix-Operator:
    $if\ \ b\ then\ p\ else\ q \equiv if\ \ b\ \ p\ \ q$

# Basis-Axiome I: Gleichheit

- Reflexivität:

$$\frac{}{t = t}\,\text{refl}$$

- Substitutivität:

$$\frac{s = t \quad P(s)}{P(t)}\,\text{subst}$$

- Extensionalität:

$$\frac{\forall\, x.fx = gx}{(\lambda\, x.fx) = (\lambda\, x.gx)}\,\text{ext}$$

- Einführungsregel:

$$\frac{}{(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q)}\ \text{iff}$$

# Basis-Axiome II: Implikation

- Einführungsregel:

$$\frac{\begin{array}{c}[P]\\ \vdots \\ Q\end{array}}{P \longrightarrow Q}\,\text{impl}$$

- Eliminationsregel:

$$\frac{P \longrightarrow Q \quad P}{Q}\,\text{mp}$$

# Die Basis-Axiome III: Auswahl

- Eliminitationsregel des Auswahloperators:

$$\frac{}{(\iota x.x = a) = a}\ \text{the\_eq}$$

- HOL ist klassisch:

$$\frac{}{(P = true) \vee (P = false)}\ \text{true\_or\_false}$$

# HOL: Die Basis-Axiome (Isabelle-Syntax)

refl : $t = t$

subst : $[\![s = t;\ P(s)]\!] \Longrightarrow P(t)$

ext : $[\![\bigwedge x.fx = gx]\!] \Longrightarrow (\lambda\, x.fx) = (\lambda\, x.gx)$

impl : $[\![P \Longrightarrow Q]\!] \Longrightarrow P \longrightarrow Q$

mp : $[\![P \longrightarrow Q;\ P]\!] \Longrightarrow Q$

iff : $(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q)$

the_eq : $(\iota x.x = a) = a$

true_or_false : $(P = true) \vee (P = false)$

# HOL: Abgeleitete Operatoren

$$
\begin{aligned}
true &\equiv (\lambda\, x.x) = (\lambda\, x.x) \\
\forall\, P &\equiv (P = \lambda\, x.true) \\
\exists\, P &\equiv \forall\, Q.(\forall\, x.Px \longrightarrow Q) \longrightarrow Q \\
false &\equiv \forall\, P.P \\
\neg P &\equiv P \longrightarrow false \\
P \wedge Q &\equiv \forall\, R.(P \longrightarrow Q \longrightarrow R) \longrightarrow R \\
P \vee Q &\equiv \forall\, R.(P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R \\
if\ P\ then\, x\ else\, y &\equiv \iota z.(P = true \longrightarrow z = x) \wedge (P = false \longrightarrow z = y)
\end{aligned}
$$

# Konservative Erweiterung

- Signatur $\Sigma = \langle T, \Omega \rangle$
  - Typdeklarationen $T$, Operationen $\Omega$
  - Definiert die Syntax: Terme $T_\Sigma$ über $\Sigma$
- Theorie $\mathcal{T}h = \langle \Sigma, \mathcal{A}x \rangle$
  - Axiome $\mathcal{A}x$
  - Theoreme: $Thm(\mathcal{T}h) \stackrel{def}{=} \{t \mid \mathcal{T}h \vdash t\}$

# Konservative Erweiterung

- Signatur $\Sigma = \langle T, \Omega \rangle$
  - Typdeklarationen $T$, Operationen $\Omega$
  - Definiert die Syntax: Terme $T_\Sigma$ über $\Sigma$
- Theorie $\mathcal{T}h = \langle \Sigma, \mathcal{A}x \rangle$
  - Axiome $\mathcal{A}x$
  - Theoreme: $Thm(\mathcal{T}h) \stackrel{def}{=} \{t \mid \mathcal{T}h \vdash t\}$
- Erweiterung: $\Sigma \subseteq \Sigma'$, $\mathcal{T}h \subseteq \mathcal{T}h'$
  - Konservativ gdw. $t \in Thm(\mathcal{T}h')$, $t \in T_\Sigma$ dann $t \in Thm(\mathcal{T}h)$
  - Keine neuen Theoreme über alte Symbole

# Konservative Erweiterung in Isabelle

- Isabelle: Konstruktion von Theorien durch konservative Erweiterungen
- Konstantendefinition (zur Signatur $\Sigma$):

  $$c :: \sigma \qquad c \equiv t$$

  - $c \notin \Sigma$
  - $t \in T_\Sigma$ (enthält nicht $c$)
  - Typvariablen in $t$ auch in $\sigma$
- Lemma: Konstantendefinition ist konservative Erweiterung
- Weitere konservative Erweiterungen: Typdefinitionen, Datentypen.

# Zusammenfassung

Logik höherer Stufe (HOL)

- Syntax basiert auf dem einfach getypten $\lambda$-Kalkül
- Drei Basis-Operatoren, Acht Basis-Axiome
- Rest folgt durch konservative Erweiterung

# Vorlesung vom 24.11.05: Grundlagen von Isabelle

# Fahrplan

- Isabelle: Systemarchitektur
- Beweis in Isabelle: Unifikation und Resolution
- Typdefinitionen
- Nützliche vordefinierte Datentypen in Isabelle

# Isabelle Systemarchitektur: LCF-Design

- Isabelle: ca. 150 Kloc SML, ca. 310 Kloc Beweise — Korrektheit?

# Isabelle Systemarchitektur: LCF-Design

- Isabelle: ca. 150 Kloc SML, ca. 310 Kloc Beweise — Korrektheit?
- Reduktion des Problems:
  - Korrektheit eines logischen Kerns
  - Rest durch Typisierung

# Isabelle Systemarchitektur: LCF-Design

- Isabelle: ca. 150 Kloc SML, ca. 310 Kloc Beweise — Korrektheit?
- Reduktion des Problems:
  - Korrektheit eines logischen Kerns
  - Rest durch Typisierung
- Abstrakter Datentyp `thm`, Inferenz-Regeln als Operationen

# Isabelle Systemarchitektur: LCF-Design

- Isabelle: ca. 150 Kloc SML, ca. 310 Kloc Beweise — Korrektheit?
- Reduktion des Problems:
  - Korrektheit eines logischen Kerns
  - Rest durch Typisierung
- Abstrakter Datentyp `thm`, Inferenz-Regeln als Operationen

```
val assume: cterm -> thm
val implies_intr: cterm -> thm -> thm
```

- Logischer Kern:
  - Typcheck, Signaturen, Unifikation, Meta-Logik: ca. 5500 LOC

# Variablen

- Meta-Variablen: können unifiziert und beliebig instantiiert werden
- freie Variablen (fixed): beliebig aber fest
- Gebundene Variablen: Name beliebig ($\alpha$-Äquivalenz)

# Variablen

- Meta-Variablen: können unifiziert und beliebig instantiiert werden
- freie Variablen (fixed): beliebig aber fest
- Gebundene Variablen: Name beliebig ($\alpha$-Äquivalenz)
- Meta-Quantoren: Isabelles Eigenvariablen

$$\frac{\bigwedge x.P(x)}{\forall x.P(x)}\,allI \qquad \texttt{!!x. P x ==> ALL x. P x}$$

  - Beliebig instantiierbar
  - Gültigkeit auf diese (Teil)-Formel begrenzt

# Formeln

- Formeln in Isabelle:

$$\frac{\phi_1, \ldots, \phi_n}{\psi} \qquad [\![\phi_1, \ldots, \phi_n]\!] \Longrightarrow \psi$$

  - $\phi_1, \ldots, \phi_n$ Formeln, $\psi$ atomar
  - Theoreme: ableitbare Formeln
  - Ableitung von Formeln: Resolution, Instantiierung, Gleichheit
- Randbemerkung:
  - $\Longrightarrow$, $\bigwedge$, $\equiv$ formen Meta-Logik
  - Einbettung anderer Logiken als HOL möglich — generischer Theorembeweiser

# Resolution

- Einfache Resolution: vorwärts (`THEN`), oder rückwärts (`rule`)
  - Achtung: Lifting von Meta-Quantoren und Bedingungen

# Resolution

- Einfache Resolution: vorwärts (`THEN`), oder rückwärts (`rule`)
  - Achtung: Lifting von Meta-Quantoren und Bedingungen
  - Randbemerkung: Unifikation höherer Stufe unentscheidbar
- Eliminationsresolution (`erule`)
- Destruktionsresolution (`drule`)

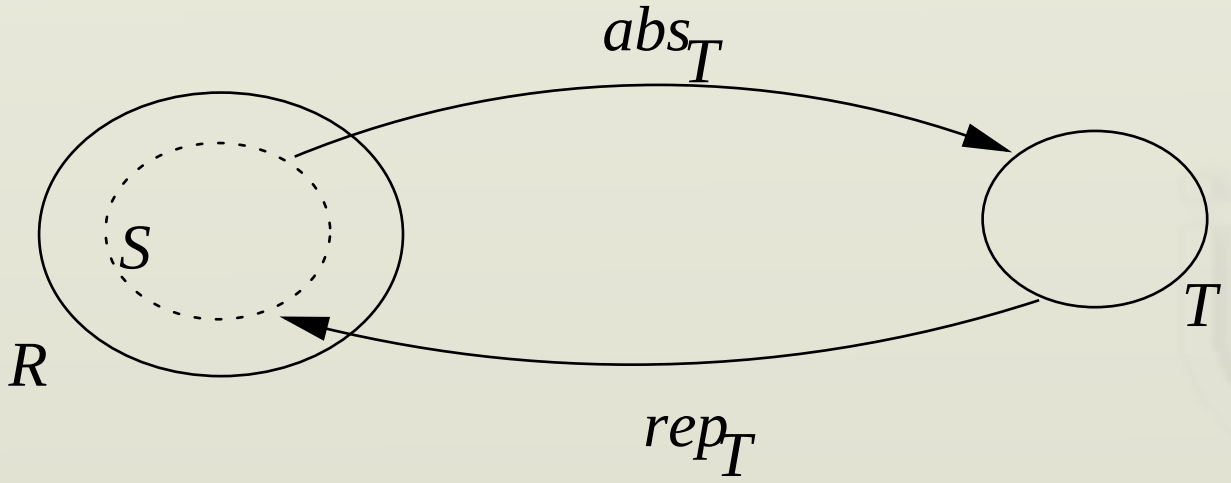
# Vorwärts oder Rückwärts?

- Vorwärtsbeweis:
  - Modifikation von Theoremen mit Attributen:
  - Resolution (`THEN`), Instantiierung (`WHERE`)
- Rückwärtsbeweis:
  - Ausgehend von Beweisziel $\psi$
  - Beweiszustand ist $[\![\phi_1, \ldots, \phi_n]\!] \Longrightarrow \psi$
  - $\phi_1, \ldots, \phi_n$: subgoals
  - Beweisverfahren: Resolution, Termersetzung, Beweissuche

# Typdefinitionen in Isabelle

Definition eines neuen Typen:

- Gegeben Typ $R$, Prädikat $S : R \Rightarrow Bool$
- Neuer Typ $T$ mit $abs_T : R \Rightarrow T, rep_T : T \Rightarrow R$ so dass
  - $S$ und $T$ isomorph:
    $\forall\, t.abs_T(rep_T t) = t, \forall\, r.Sr \longrightarrow rep_T(abs_T r) = r$
  - $T$ nicht leer: $\exists\, t.St$



- Lemma: Typdefinitionen sind konservative Erweiterungen.

# Beispiel: Produkte

- Ausgangstyp $R_{\alpha,\beta} \equiv \alpha \Rightarrow \beta \Rightarrow Bool$
- Neuer Typ $T_{\alpha,\beta} \equiv \alpha \times \beta$
  - Idee: Prädikat $p : \alpha \Rightarrow \beta \Rightarrow Bool$ repräsentiert $(a, b)$
    falls $p(x, y) = true \longleftrightarrow x = a \wedge y = b$
- $S = \lambda f . \exists\, a\ b . f = \lambda\, x\ y . x = a \wedge y = b$
- Abstraktionsfunktion: $abs_{\times} p = \iota a\ b . p\ a\ b$
- Repräsentationsfunktion: $rep_{\times}(a, b) = p$ mit
  $p(x, y) = true \longleftrightarrow x = a \wedge y = b$

# Nützliche Typen: natürliche Zahlen und algebraische Datentypen

- Natürliche Zahlen: nicht konservativ!
- Erfordert zusätzliches Axiom: Unendlichkeit.
- Neuer Typ $ind$mit $Z :: ind, S : ind \Rightarrow S$ und

$$inj\, S \quad Sx \neq Z$$

- Damit `nat` definierbar.
- Warum $ind$? Auch für andere Datentypen. . .
- Damit algebraische Datentypen (`datatype`) etc.
  - `datatype`: nur kovariante Rekursion

# Einige nützliche Typen

- Theorie `Set`: getypte Mengen, $\alpha$ `set` $\equiv \alpha \Rightarrow Bool$
- Theorien `Sum` und `Produkt`: Summe und Produkte
- Numerische Typen:
  - Natürliche Zahlen `nat`, ganze Zahlen `int`, reele Zahlen `real`, komplexe Zahlen `complex`.
  - Arithmetische Operatoren `+`, `-`, `*`; Konstanten ("12343423");
  - Entscheidungsprozeduren
- Theorie `Datatype`: $\alpha$ `option` ist entweder `Some x` oder `None`
- Theorie `List`: Listen $\alpha$`list`
- Theorie `Map`: endliche, partielle Abbildungen $\alpha$ `~=>` $\beta$

# Zusammenfassung

- Isabelle: LCF-Architektur mit logischem Kern und Meta-Logik
- Beweisen durch Resolution
  - Lifting; Eliminations- und Destruktionsresolution.
- Typdefinitionen: Einschränkung bestehender Typen
- Der Typ $ind$: Unendlichkeitsaxiom
- Nützliche vordefinierte Typen
  - numerische Typen, Listen, Abbildungen, Mengen. . .

# Vorlesung vom 05.12.05: Modellierung funktionaler Programme

# Wo sind wir?

- Teil I: Grundlagen formaler Logik
- Teil II: Modellierung
  - Datentypen
  - Rekursive Programme
  - Imperative Programme
- Teil III: Spezifikationsformalismen

# Fahrplan

- Funktionale Programme:
  - Rekursive Funktionen + algebraische Datentypen
- Rekursion auch bei imperative Programme
- Modellierung von Rekursion durch Fixpunkte
- Beweis durch Fixpunktinduktion

# Rekursion

- Problem: rekursive Funktionen

$$fac \equiv \lambda\, n.\, if\ \ n = 0\ \ then\, 1\ else\, n * (fac\ (n-1))$$

  - Keine konservative Definition

# Rekursion

- Problem: rekursive Funktionen

$$fac \equiv \lambda\, n.\, if\ \ n = 0\ \ then\, 1\ else\, n * (fac\ (n-1))$$

  - Keine konservative Definition
- Lösung: Modellierung durch Fixpunkte
  - Gegeben $fix : (\alpha \Rightarrow \alpha) \Rightarrow \alpha$ mit $fix\ F = F(fix\ F)$, dann

$$fac \equiv fix(\lambda\, F(\lambda\, n.\, if\ \ n = 0\ \ then\, 1\ else\, n * (F\ (n-1))))$$

    $F$ heißt Approximationsfunktional
  - Typ von $fix$? Hier: $fix :: (Nat \Rightarrow Nat) \Rightarrow Nat \Rightarrow Nat)$
  - Wie definieren wir $fix$?

# Fixpunkte

- Ungetypter Lambda-Kalkül: Fixpunktkombinator $Y$
- Getypter Lambda-Kalkül: Familie von Fixpunktkombinatoren $Y_\alpha : (\alpha \Rightarrow \alpha) \Rightarrow \alpha$
- In HOL: nicht jede Funktion hat einen Fixpunkt.

# Fixpunkte

- Ungetypter Lambda-Kalkül: Fixpunktkombinator $Y$
- Getypter Lambda-Kalkül: Familie von Fixpunktkombinatoren $Y_\alpha : (\alpha \Rightarrow \alpha) \Rightarrow \alpha$
- In HOL: nicht jede Funktion hat einen Fixpunkt. (Gegenbeispiel: $\neg$)
- Lösung: Konstruktion von Fixpunkten
- Verschiedene Ansätze
  - Allgemein: Ordnung, aufsteigende Ketten
  - Vollständige Halbordnungen (LCF)
  - Wohlfundierte Rekursion

# Vollständige Halbordnungen

Def (Halbordung): reflexive, transitive, antisymmetrische Relation $A \times A$.

Def (Vollständige Halbordnung, cpo): Halbordnung $(A, \sqsubseteq)$ mit kleinstem Element $\bot \in A$, so dass jede aufsteigende Kette $F$ ein Supremum $sup\, F$ besitzt.

(Gerichtete volllständige Halbordnung, dcpo: ohne kleinstes Element)

Beispiele:

- Für Menge $X$ ist $(X_\bot, \sqsubseteq)$ flacher cpo mit $X_\bot = X + \{\bot\}$, $x \sqsubseteq y$ gdw. $x = \bot$.
- Für cpo's $A, B$ ist Funktionsraum $A \Rightarrow B$ ein cpo unter punktweiser Ordnung: $f \sqsubseteq g$ gdw. $\forall\, a \in A. fa \sqsubseteq ga$

# Stetigkeit und Fixpunkte

Def: Funktion $f : A \Rightarrow B$ ($A, B$ cpos) heißt

- monoton, wenn $a \sqsubseteq a' \longrightarrow f(a) \sqsubseteq f(a')$.
- stetig, wenn $f$ monoton und $f(sup\, F) = sup\, fF$

Fixpunktsatz: Jede Stetige Funktion $f : A \Rightarrow A$ hat kleinsten Fixpunkt $fix\, f \in A$.

# Logik berechenbarer Funktionen

- Alle Basisdatentypen ($Nat$, $Bool$ etc.) sind flache cpo's.

# Logik berechenbarer Funktionen

- Alle Basisdatentypen ($Nat$, $Bool$ etc.) sind flache cpo's.
- Basisfunktionen ($if\ then\ else, +, -$ etc.) sind geliftet (und damit stetig).

# Logik berechenbarer Funktionen

- Alle Basisdatentypen ($Nat$, $Bool$ etc.) sind flache cpo's.
- Basisfunktionen ($if\ then\ else, +, -$ etc.) sind geliftet (und damit stetig).
- Funktionsräume $A \Rightarrow B$ werden punktweise geordnet.

# Logik berechenbarer Funktionen

- Alle Basisdatentypen ($Nat$, $Bool$ etc.) sind flache cpo's.
- Basisfunktionen ($if\ then\ else, +, -$ etc.) sind geliftet (und damit stetig).
- Funktionsräume $A \Rightarrow B$ werden punktweise geordnet.
- Approximationsfunktional $F$ ist stetig, wenn es aus stetigen Funktionen besteht.
- Laut Fixpunktsatz hat jedes stetige Approximationsfunktional einen kleinsten Fixpunkt.

# LCF in HOL

- Kodierung von LCF in HOL: HOLCF
  - Neue Objektlogik
  - Konservative Entwicklung
  - Typklasse cpo, stetige Funktionen, Fixpunkte
- Problem:
  - keine automatische $\beta$-Reduktion
  - Stetigkeitsbeweise nötig, neue Datentypen
  - Schlechtere Beweisunterstützung (als in HOL)
- Alternativer Ansatz in HOL: wohlfundierte Rekursion

# Wohlfundierte Ordnung und Induktion

- Def: Ordnung $\sqsubseteq \subseteq A \times A$ wohlfundiert: keine unendlich absteigenden Ketten $x_0, x_2, x_3, \ldots$ mit $x_{n+1} \sqsubseteq x_n$.
- Def (Wohlfundierte Induktion): Wenn $\forall\, x.(\forall\, y.y \sqsubseteq x \wedge P(y)) \longrightarrow P(x)$, dann $\forall\, x.P(x)$.
  - Induktionsannahme gilt für alle kleineren Werte
  - Keine explizite Induktionsverankerung

# Wohlfundierte Ordnung und Induktion

- Def: Ordnung $\sqsubseteq \subseteq A \times A$ wohlfundiert: keine unendlich absteigenden Ketten $x_0, x_2, x_3, \ldots$ mit $x_{n+1} \sqsubseteq x_n$.
- Def (Wohlfundierte Induktion): Wenn
  $\forall\, x.(\forall\, y.y \sqsubseteq x \wedge P(y)) \longrightarrow P(x)$, dann $\forall\, x.P(x)$.
  - Induktionsannahme gilt für alle kleineren Werte
  - Keine explizite Induktionsverankerung
- Wohlfundiertheit in HOL:

$$wf :: (\alpha \times \alpha)\ set \Rightarrow Bool$$
$$wf\ R \equiv (\forall\, x.(\forall\, y.y \sqsubseteq x \longrightarrow P(y)) \longrightarrow P(x)) \longrightarrow \forall\, x.P(x)$$

# Wohlfundierte Rekursion

- Für $f : \alpha \Rightarrow \beta$
- Wohlfundierte Ordnung auf $\alpha$ Parameter der Definition
- Voraussetzung: rekursiver Aufruf mit kleineren Argumenten.

  D.h. für Approximationsfunktional $H : (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$ muß gelten $Hfa = Hf'a$ gdw. $\forall\, x \sqsubseteq a.fx = f'x$

# Wohlfundierte Rekursion

- Gegeben $H : (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$ und $\sqsubseteq$.
- Ziel: Definition des Fixpunktes als Relation $\alpha \times \beta \; Set$
- Wohlfundierte Rekursionsordnung:
  Definiere Relation $R_{\sqsubseteq,H}$ als kleinste Relation so dass
  $\forall\, z.(z \sqsubseteq x) \longrightarrow (z, gz) \in R_{\sqsubseteq,H}$, dann $(x, Hgx) \in R_{\sqsubseteq,H}$.
  - Induktive Definition
- Approximationsfunktional $H$ bei $x$ 'abschneiden' (`cut`):

$$f\,|_x\,(y) \stackrel{def}{=} \begin{cases} f(x) & y \sqsubseteq x \\ undefined & ows \end{cases}$$

# Wohlfundierte Rekursion

- Gegeben Approximationsfunktional $H : (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$ und $\sqsubseteq$ auf $\alpha$.
- Rekursionsordnung $R_{F,\sqsubseteq}$ wird induktiv als Relation auf $\alpha \times \beta$ definiert mit $F \stackrel{def}{=} \lambda fx.H(f\ |_x)x$.
- Wohlfundierte Rekursion gegeben durch $R_{F,\sqsubseteq}$:

  $wfrec_{\sqsubseteq}\, H = \iota y.(x, y) \in R_{\lambda fx.H(f|_x)x,\sqsubseteq}$

- Wohldefiniert, wenn jeder Aufruf Argumente verkleinert

# Wohlfundierte Rekursion in Isabelle

- Wohlfundierte Ordnung als Argument der Funktionsdefinition
- Wohldefiniertheit als Beweisverpflichtung
- Vordefinierte Taktiken und Ordnungen: `less_than`, `measure`, . . .
- 1. Beispiel: Fakultät

```
consts
  fac :: "nat => nat"
recdef fac "less_than"
 "fac n = (if n= 0 then 1 else n* fac (n - 1))"
```

- 2. Beispiel: Mergesort . . .

```
consts merge :: "('a::linorder)list * 'a list => 'a list"
recdef merge "measure (%(xs,ys). size xs + size ys)"
  "merge(x#xs, y#ys) =
     (if x < y then x # merge(xs, y#ys)
               else y # merge(x#xs, ys))"
  "merge(xs,[]) = xs"
  "merge([],ys) = ys"

consts msort :: "('a::linorder) list => 'a list"
recdef msort "measure size"
    "msort [] = []"
    "msort [x] = [x]"
    "msort xs = merge(msort(take (size xs div 2) xs),
      msort(drop (size xs div 2) xs))"
```

# Datentypen

- Algebraische Datentypen: auch hier ist Rekursion ein Fixpunkt.

```
datatype 'a list = Nil | Cons 'a "'a * 'a list"

type 'a list = fix M. Nil | Cons 'a "'a * M"
```

- Kleinster Fixpunkt: endliche Datentypen
  - Zum Beispiel Listen
- Größter Fixpunkt: unendliche Datentypen
  - Ströme, unendliche Listen:
    `type 'a stream = fix M. Cons 'a "'a * 'a M"`

# Datentypen als Fixpunkte

- In HOL
  - Nur kovariante Rekursion
  - Lösung ist kleinster Fixpunkt
- Mit vollständigen Halbordnungen als Trägermengen (HOLCF):
  - Jede rekursive Typgleichung (domain equation) hat Lösung.
  - Kleinster Fixpunkt auch größter Fixpunkt
  - Damit auch unendliche Datentypen möglich (e.g. Listen)

# Zusammenfassung

- Modellierung rekursiver Funktionen durch Fixpunkte
- Problem: Nicht jede Funktion hat einen Fixpunkt
- Lösungen:
  - Erweiterung der Logik: stetige Funktionen auf vollständigen Halbordnungen
  - Vorteil: beliebige Rekursion, Nachteil: neue Logik, umständlicheres Beweisen
  - Beschränkung der Rekursion: wohlfundierte Rekursion
  - Vorteil: gute Beweisunterstützung, Nachteil: nicht allgemein, Definition umständlich

# Vorlesung vom 12.12.05: Modellierung imperativer Programme

# Wo sind wir?

- Teil I: Grundlagen formaler Logik
- Teil II: Modellierung
  - Datentypen
  - Rekursive Programme
  - Imperative Programme
- Teil III: Spezifikationsformalismen

# Fahrplan

- Nachtrag:
  - Datentypen als Fixpunkte
- Imperative Programme:
  - Ausführung ist Zustandstransformation
  - **IMP** — eine einfache imperative Sprache
  - Denotationale und operationale Semantik

# Die Sprache IMP

- Zahlen: **N** $= \{0, 1, 2, 3, \ldots\}$
- Wahrheitswerte: **T** $= \{\texttt{True}, \texttt{False}\}$
- Variablen: **Loc**
- Arithmetische Ausdrücke: $a \in$ **Aexp**

  $a ::= n \mid x \mid a \texttt{ + } a \mid a \texttt{ - } a \mid a \texttt{ * } a$ $\qquad n \in$ **N**$, x \in$ **Loc**
- Boolsche Ausdrücke: $b \in$ **Bexp**

  $b ::= v \mid a \texttt{ = } a \mid a \texttt{ <= } a \mid \texttt{not } b \mid b \texttt{ and } b \mid b \texttt{ or } b$ $\qquad v \in$ **T**
- Anweisungen:

  **Com**$c ::= \texttt{skip} \mid x := a \mid c; c \mid$
  $\texttt{if } b \texttt{ then } c \texttt{ else } c \texttt{ fi} \mid \texttt{while } b \texttt{ do } c \texttt{ od}$

# Beispielprogramme in IMP

$Fac \equiv$

```
M := 1;
while 0 < N do
  M:= N* M;
  N:= N- 1
od
```

$Euclid \equiv$

```
while (not (M= N)) do
  if (M<= N)
  then N:= N- M
  else M:= M- N
  fi
od
```

# Semantik

- Denotationale Semantik:
  - Abbildung $\mathcal{D} : \mathbf{Com} \to \mathcal{S}$ in semantischen Bereich
  - Für: Programmentwicklung und Programmtransformation
- Operationale Semantik:
  - Syntaxgesteuerte Ableitungsregeln (Zustandsübergang)
  - Für: Sprachdefinition, Compilerverifikation
- Axiomatische Semantik:
  - Vor- und Nachbedingungen für jede Anweisung
  - Für: Programmverifikation

# Denotationale Semantik

- Letzte VL: denotationelle Semantik einer funktionalen Sprache
  - Programm als Funktion $\mathbf{cpo} \to \mathbf{cpo}$
  - Flache Einbettung

# Denotationale Semantik

- Letzte VL: denotationelle Semantik einer funktionalen Sprache
  - Programm als Funktion $\mathbf{cpo} \to \mathbf{cpo}$
  - Flache Einbettung
- Hier: denotationelle Semantik für **IMP**
  - Programm als Funktion zwischen Systemzuständen
  - Tiefe Einbettung

# Denotationale Semantik

- Letzte VL: denotationelle Semantik einer funktionalen Sprache
  - Programm als Funktion $\mathbf{cpo} \to \mathbf{cpo}$
  - Flache Einbettung
- Hier: denotationelle Semantik für **IMP**
  - Programm als Funktion zwischen Systemzuständen
  - Tiefe Einbettung
- Systemzustand: $\Sigma \stackrel{def}{=} \mathbf{Loc} \rightharpoonup \mathbb{N}$
- Semantische Funktionen:
  - Arithmetische Ausdrücke: $\mathcal{E} : \mathbf{Aexp} \to (\Sigma \to \mathbb{N})$
  - Boolsche Ausdrücke: $\mathcal{B} : \mathbf{Bexp} \to (\Sigma \to \mathit{Bool})$
  - Anweisungen: $\mathcal{D} : \mathbf{Com} \to (\Sigma \rightharpoonup \Sigma)$

# Denotationale Semantik für IMP: Anweisungen

$$\mathcal{D}[\![\texttt{skip}]\!] \stackrel{def}{=} \lambda\,\sigma^{\Sigma}.\sigma$$

$$\mathcal{D}[\![X := a]\!] \stackrel{def}{=} \lambda\,\sigma^{\Sigma}.\sigma[\mathcal{E}[\![a]\!]\sigma/X]$$

$$\mathcal{D}[\![c_0;\ c_1]\!] \stackrel{def}{=} \mathcal{D}[\![c_1]\!] \circ \mathcal{D}[\![c_0]\!]$$

$$\mathcal{D}[\![\texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1\ \texttt{fi}]\!] \stackrel{def}{=} \lambda\,\sigma^{\Sigma}.\begin{cases} \mathcal{D}[\![c_0]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = true \\ \mathcal{D}[\![c_1]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = false \end{cases}$$

$$\mathcal{D}[\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}]\!] \stackrel{def}{=} fix\ \Gamma$$

$$\text{mit } \Gamma(\phi) \stackrel{def}{=} \lambda\,\sigma^{\Sigma}.\begin{cases} \phi \circ \mathcal{D}[\![c]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = true \\ \sigma & \mathcal{B}[\![b]\!]\sigma = false \end{cases}$$

# Strukturelle Operationale Semantik

- Syntaxgesteuerte Auswertungsregeln
- Relativ zu einem Systemzustand
- Relation $\Rightarrow$ auf Konfigurationen: $\langle a, \sigma \rangle$ mit $a \in \mathbf{Aexp}, \mathbf{Bexp}, \mathbf{Com}$
  - Eigentlich Familie von Relationen $\{\Rightarrow_i\}_{i \in \mathbf{Aexp},\mathbf{Bexp},\mathbf{Com}}$
- Regeln der Form

$$\frac{\langle a_1, \sigma \rangle \Rightarrow \langle a_1', \sigma_n' \rangle \quad \ldots \quad \langle a_n, \sigma \rangle \Rightarrow \langle a_n', \sigma_n' \rangle}{\langle a, \sigma \rangle \Rightarrow \langle a', \sigma' \rangle} \, C$$

  ggf. mit Randbedingung $C$

# Operationale Semantik

- Beispielregeln für **Aexp** und **Bexp**:

$$\frac{}{\langle \mathtt{X}, \sigma \rangle \Rightarrow \sigma(\mathtt{X})} \qquad \frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \mathbf{+} a_1, \sigma \rangle \Rightarrow n + m}$$

# Operationale Semantik

- Beispielregeln für **Aexp** und **Bexp**:

$$\frac{}{\langle \mathtt{X}, \sigma \rangle \Rightarrow \sigma(\mathtt{X})}$$

$$\frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \mathtt{+} a_1, \sigma \rangle \Rightarrow n + m}$$

$$\frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \mathtt{=} a_1, \sigma \rangle \Rightarrow \mathit{true}} \; n = m$$

$$\frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \mathtt{=} a_1, \sigma \rangle \Rightarrow \mathit{false}} \; n \neq m$$

# Operationale Semantik

- Beispielregeln für **Aexp** und **Bexp**:

$$\frac{}{\langle \texttt{X}, \sigma \rangle \Rightarrow \sigma(\texttt{X})} \qquad \frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \texttt{ + } a_1, \sigma \rangle \Rightarrow n + m}$$

$$\frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \texttt{ = } a_1, \sigma \rangle \Rightarrow \mathit{true}} \; n = m \qquad \frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \texttt{ = } a_1, \sigma \rangle \Rightarrow \mathit{false}} \; n \neq m$$

- Beispielregeln für Anweisungen:

$$\frac{\langle a, \sigma \rangle \Rightarrow n}{\langle X := a, \sigma \rangle \Rightarrow \sigma[n/X]} \qquad \frac{\langle b, \sigma \rangle \Rightarrow \mathit{false}}{\langle \texttt{while } b \texttt{ do } c \texttt{ od}, \sigma \rangle \Rightarrow \sigma}$$

$$\frac{\langle b, \sigma \rangle \Rightarrow \mathit{true} \quad \langle c, \sigma \rangle \Rightarrow \tau' \quad \langle \texttt{while } b \texttt{ do } c \texttt{ od}, \tau' \rangle \Rightarrow \tau}{\langle \texttt{while } b \texttt{ do } c \texttt{ od}, \sigma \rangle \Rightarrow \tau}$$

# Äquivalenz der Semantiken

Lemma: Für $a \in$ **Aexp**, $n \in \mathbb{N}$, $\mathcal{E}[\![a]\!]\sigma = n \longleftrightarrow \langle a, \sigma\rangle \Rightarrow n$
Beweis: Strukturelle Induktion über $a$

Lemma: Für $b \in$ **BExp**, $t \in Bool$, $\mathcal{B}[\![b]\!]\sigma = t \longleftrightarrow \langle b, \sigma\rangle \Rightarrow t$
Beweis: Strukturelle Induktion über $b$

Lemma: Für $c \in$ **Com**, $\langle c, \sigma\rangle \Rightarrow \sigma' \longrightarrow \mathcal{D}[\![c]\!]\sigma = \sigma'$
Beweis: Induktion über der Herleitung von $\langle c, \sigma\rangle \Rightarrow \sigma'$.

Satz (Äquivalenz der Semantiken): Für $c \in$ **Com**, und $\sigma, \sigma' \in \Sigma$,

$$\langle c, \sigma\rangle \Rightarrow \sigma' \longleftrightarrow \mathcal{D}[\![c]\!]\sigma = \sigma'$$

Beweis: Strukturelle Induktion über $c$ mit Fixpunktinduktion für `while`

# Axiomatische Semantik

- Abstraktionsgrad zwischen operationaler und denotationaler Semantik
- Idee: Annotierte Programme

  `M:= 1;`$\{M = 1, N = n\}$

  `while 0 < N do M:= N* M; N:= N- 1 od`$\{N \leq 0 \wedge M = n!\}$
- Vor- und Nachbedingungen: $\{A\}c\{B\}$

$$\frac{\{A \wedge b\}c\{A\}}{\{A\}\texttt{while } b \texttt{ do } c \texttt{ od}\{A \wedge \neg b\}}$$

- Partielle und totale Korrektheit.
- Hoare-Kalkül, schwächste Vorbedingung $\rightsquigarrow$ nächtes Jahr

# Zusammenfassung

- Semantik von imperative Programme: Zustandsübergang
  - Denotationale und operationale Semantik
- Spezifikation imperativer Programme: welche Zustandsübergänge?
- Nächstes Jahr: Hoare-Kalkül, schwächste Vorbedingung; Z.

# Anhang

Auf den folgenden Seiten:

- Vollständige operationale Semantik für **IMP**:
  - Arithmetische Ausdrücke
  - Boolsche Ausdrücke (zwei Varianten)
  - Anweisungen
- Vollständige Denotationale Semantik für **IMP**:
  - Arithmetische Ausdrücke
  - Boolsche Ausdrücke

# Operationale Semantik I: Aexp

Zahlen:
$$\frac{}{\langle n, \sigma \rangle \Rightarrow n}$$

Variablen:
$$\frac{}{\langle \mathtt{X}, \sigma \rangle \Rightarrow \sigma(\mathtt{X})}$$

Addition:
$$\frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \mathtt{+} a_1, \sigma \rangle \Rightarrow n + m}$$

Subtraktion:
$$\frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \mathtt{-} a_1, \sigma \rangle \Rightarrow n - m}$$

Multiplikation:
$$\frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \mathtt{*} a_1, \sigma \rangle \Rightarrow n \cdot m}$$

# Operationale Semantik II: Bexp

$$\frac{}{\langle \texttt{True}, \sigma \rangle \Rightarrow true} \qquad \frac{}{\langle \texttt{False}, \sigma \rangle \Rightarrow false}$$

$$\frac{\langle b, \sigma \rangle \Rightarrow false}{\langle \texttt{not}\ \ b, \sigma \rangle \Rightarrow true} \qquad \frac{\langle b, \sigma \rangle \Rightarrow true}{\langle \texttt{not}\ \ b, \sigma \rangle \Rightarrow false}$$

$$\frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \texttt{ = } a_1, \sigma \rangle \Rightarrow true} \text{ if } n = m \qquad \frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \texttt{ = } a_1, \sigma \rangle \Rightarrow false} \text{ if } n \neq m$$

$$\frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \texttt{ <= } a_1, \sigma \rangle \Rightarrow true} \text{ if } n \leq m \qquad \frac{\langle a_0, \sigma \rangle \Rightarrow n \quad \langle a_1, \sigma \rangle \Rightarrow m}{\langle a_0 \texttt{ <= } a_1, \sigma \rangle \Rightarrow false} \text{ if } n > m$$

# Operationale Semantik IIa: Bexp (sequentielles `and`, `or`)

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{false}}{\langle b_0 \texttt{ and } b_1, \sigma\rangle \Rightarrow \mathit{false}}$$

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{true} \quad \langle b_1, \sigma\rangle \Rightarrow \mathit{false}}{\langle b_0 \texttt{ and } b_1, \sigma\rangle \Rightarrow \mathit{false}}$$

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{true} \quad \langle b_1, \sigma\rangle \Rightarrow \mathit{true}}{\langle b_0 \texttt{ and } b_1, \sigma\rangle \Rightarrow \mathit{true}}$$

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{true}}{\langle b_0 \texttt{ or } b_1, \sigma\rangle \Rightarrow \mathit{true}}$$

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{false} \quad \langle b_1, \sigma\rangle \Rightarrow \mathit{true}}{\langle b_0 \texttt{ or } b_1, \sigma\rangle \Rightarrow \mathit{true}}$$

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{false} \quad \langle b_1, \sigma\rangle \Rightarrow \mathit{false}}{\langle b_0 \texttt{ or } b_1, \sigma\rangle \Rightarrow \mathit{false}}$$

# Operationale Semantik IIb: Bexp (paralleles `and`,`or`)

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{false} \quad \langle b_1, \sigma\rangle \Rightarrow \mathit{false}}{\langle b_0 \texttt{ and } b_1, \sigma\rangle \Rightarrow \mathit{false}}$$

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{false} \quad \langle b_1, \sigma\rangle \Rightarrow \mathit{true}}{\langle b_0 \texttt{ and } b_1, \sigma\rangle \Rightarrow \mathit{false}}$$

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{true} \quad \langle b_1, \sigma\rangle \Rightarrow \mathit{false}}{\langle b_0 \texttt{ and } b_1, \sigma\rangle \Rightarrow \mathit{false}}$$

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{true} \quad \langle b_1, \sigma\rangle \Rightarrow \mathit{true}}{\langle b_0 \texttt{ and } b_1, \sigma\rangle \Rightarrow \mathit{true}}$$

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{true} \quad \langle b_1, \sigma\rangle \Rightarrow \mathit{true}}{\langle b_0 \texttt{ or } b_1, \sigma\rangle \Rightarrow \mathit{true}}$$

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{true} \quad \langle b_1, \sigma\rangle \Rightarrow \mathit{false}}{\langle b_0 \texttt{ or } b_1, \sigma\rangle \Rightarrow \mathit{true}}$$

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{false} \quad \langle b_1, \sigma\rangle \Rightarrow \mathit{true}}{\langle b_0 \texttt{ or } b_1, \sigma\rangle \Rightarrow \mathit{true}}$$

$$\frac{\langle b_0, \sigma\rangle \Rightarrow \mathit{false} \quad \langle b_1, \sigma\rangle \Rightarrow \mathit{false}}{\langle b_0 \texttt{ or } b_1, \sigma\rangle \Rightarrow \mathit{false}}$$

# Operationale Semantik III: Com

$$\frac{}{\langle \texttt{skip}, \sigma \rangle \Rightarrow \sigma} \qquad \frac{\langle a, \sigma \rangle \Rightarrow n}{\langle X := a, \sigma \rangle \Rightarrow \sigma[n/X]}$$

$$\frac{\langle b, \sigma \rangle \Rightarrow \mathit{true} \quad \langle c_0, \sigma \rangle \Rightarrow \tau}{\langle \texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1\ \texttt{fi}, \sigma \rangle \Rightarrow \tau} \qquad \frac{\langle b, \sigma \rangle \Rightarrow \mathit{false} \quad \langle c_1, \sigma \rangle \Rightarrow \tau}{\langle \texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1\ \texttt{fi}, \sigma \rangle \Rightarrow \tau}$$

$$\frac{\langle b, \sigma \rangle \Rightarrow \mathit{false}}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma \rangle \Rightarrow \sigma} \qquad \frac{\langle c_0, \sigma \rangle \Rightarrow \tau \quad \langle c_1, \tau \rangle \Rightarrow \tau'}{\langle c_0;\ c_1, \sigma \rangle \Rightarrow \tau'}$$

$$\frac{\langle b, \sigma \rangle \Rightarrow \mathit{true} \quad \langle c, \sigma \rangle \Rightarrow \tau' \quad \langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \tau' \rangle \Rightarrow \tau}{\langle \texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}, \sigma \rangle \Rightarrow \tau}$$

# Denotationale Semantik I: Aexp

$$
\begin{aligned}
\mathcal{E}[\![n]\!] &\stackrel{def}{=} \lambda\,\sigma^{\Sigma}.n \\
\mathcal{E}[\![X]\!] &\stackrel{def}{=} \lambda\,\sigma^{\Sigma}.\sigma(X) \\
\mathcal{E}[\![a_0 \texttt{ + } a_1]\!] &\stackrel{def}{=} \lambda\,\sigma^{\Sigma}.(\mathcal{E}[\![a_0]\!]\sigma + \mathcal{E}[\![a_1]\!]\sigma) \\
\mathcal{E}[\![a_0 \texttt{ - } a_1]\!] &\stackrel{def}{=} \lambda\,\sigma^{\Sigma}.(\mathcal{E}[\![a_0]\!]\sigma - \mathcal{E}[\![a_1]\!]\sigma) \\
\mathcal{E}[\![a_0 \texttt{ * } a_1]\!] &\stackrel{def}{=} \lambda\,\sigma^{\Sigma}.(\mathcal{E}[\![a_0]\!]\sigma \cdot \mathcal{E}[\![a_1]\!]\sigma)
\end{aligned}
$$

# Denotationale Semantik II: Bexp

$$
\begin{aligned}
\mathcal{B}[\![\texttt{True}]\!] &\stackrel{def}{=} \lambda\sigma^{\Sigma}.true \\
\mathcal{B}[\![\texttt{False}]\!] &\stackrel{def}{=} \lambda\sigma^{\Sigma}.false \\
\mathcal{B}[\![\texttt{not } b]\!] &\stackrel{def}{=} \lambda\sigma^{\Sigma}.\neg\mathcal{B}[\![b]\!]\sigma \\
\mathcal{B}[\![a_0 \texttt{ = } a_1]\!] &\stackrel{def}{=} \lambda\sigma^{\Sigma}.\begin{cases} true & \mathcal{E}[\![a_0]\!]\sigma = \mathcal{E}[\![a_1]\!]\sigma \\ false & \mathcal{E}[\![a_0]\!]\sigma \neq \mathcal{E}[\![a_1]\!]\sigma \end{cases} \\
\mathcal{B}[\![a_0 \texttt{ <= } a_1]\!] &\stackrel{def}{=} \lambda\sigma^{\Sigma}.\begin{cases} true & \mathcal{E}[\![a_0]\!]\sigma \leq \mathcal{E}[\![a_1]\!]\sigma \\ false & \mathcal{E}[\![a_0]\!]\sigma > \mathcal{E}[\![a_1]\!]\sigma \end{cases} \\
\mathcal{B}[\![b_0 \texttt{ and } b_1]\!] &\stackrel{def}{=} \lambda\sigma^{\Sigma}.\mathcal{B}[\![b_0]\!]\sigma \wedge \mathcal{B}[\![b_1]\!]\sigma \\
\mathcal{B}[\![b_0 \texttt{ or } b_1]\!] &\stackrel{def}{=} \lambda\sigma^{\Sigma}.\mathcal{B}[\![b_0]\!]\sigma \vee \mathcal{B}[\![b_1]\!]\sigma
\end{aligned}
$$

# Denotationale Semantik für IMP: Anweisungen

$$\mathcal{D}[\![\texttt{skip}]\!] \stackrel{def}{=} \lambda\,\sigma^{\Sigma}.\sigma$$

$$\mathcal{D}[\![X := a]\!] \stackrel{def}{=} \lambda\,\sigma^{\Sigma}.\sigma[\mathcal{E}[\![a]\!]\sigma/X]$$

$$\mathcal{D}[\![c_0;\ c_1]\!] \stackrel{def}{=} \mathcal{D}[\![c_1]\!] \circ \mathcal{D}[\![c_0]\!]$$

$$\mathcal{D}[\![\texttt{if}\ b\ \texttt{then}\ c_0\ \texttt{else}\ c_1\ \texttt{fi}]\!] \stackrel{def}{=} \lambda\,\sigma^{\Sigma}.\begin{cases} \mathcal{D}[\![c_0]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = true \\ \mathcal{D}[\![c_1]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = false \end{cases}$$

$$\mathcal{D}[\![\texttt{while}\ b\ \texttt{do}\ c\ \texttt{od}]\!] \stackrel{def}{=} fix\ \Gamma$$

$$\text{mit } \Gamma(\phi) \stackrel{def}{=} \lambda\,\sigma^{\Sigma}.\begin{cases} \phi \circ \mathcal{D}[\![c]\!]\sigma & \mathcal{B}[\![b]\!]\sigma = true \\ \sigma & \mathcal{B}[\![b]\!]\sigma = false \end{cases}$$

# Vorlesung vom 09.01.06: Algebraische Spezifikationen

# Wo sind wir?

- Teil I: Grundlagen formaler Logik
- Teil II: Modellierung
- Teil III: Spezifikationsformalismen
  - Algebraische Spezifikation mit Isabelle und CASL
  - Modellbasierte Spezifikation mit Z

# Algebraische Spezfikation

- Idee: Spezifikation ist Signatur, Programme sind Algebren
- Mathematische Grundlage: universelle Algebra
- Geschichtliches:
  - Entstanden um 1976 (ADJ-Gruppe)
  - In den 80ern Vielzahl von algebraischen Sprachen
  - Ende 90er Entwicklung der Einheitssprache CASL

# Algebraische Spezifikation: die Grundidee

- Deklaration von Typen und Operationen in Signatur
- Gewünschte Eigenschaften als Axiome
- Semantik: lose (alle Algebren) vs. initial (Termalgebra)
  - Aber: Semantik ist etwas für verregnete Herbsttage
  - Hier: Beweis und (syntaktische) Herleitung
- Implementationsbegriff durch Signatur- und Theoriemorphismen

# Ein klassisches Beispiel

- Ein Stack hat zwei Sorten und vier Operationen:

```
typedecl 'a stack
consts
  empty :: "'a stack"
  push  :: "'a stack => 'a => 'a stack"
  pop   :: "'a stack => 'a stack"
  top   :: "'a stack => 'a"
```

- Axiome: `pop` und `top` invers zu `push`

```
axioms a1: "top (push s x) = x"
       a2: "pop (push s x) = s"
```

  - `pop`, `top` partiell?
  - Keine Seiteneffekte

# Typen

Def (Typdeklarationen): $\mathcal{T} = \langle T, tar \rangle$, wobei $T$ Menge von Typkonstanten, mit $tar : T \to \mathbb{N}$ Typaritäten

Def: Aus Typdeklaration $\mathcal{T}$ generierte Typen $\mathcal{T}^*$ ist kleinste Menge

(i) $\alpha \in \mathcal{V}_{\mathcal{T}ype} \Longrightarrow \alpha \in \mathcal{T}^*$

(ii) $t_1, \ldots, t_n \in \mathcal{T}^*, c \in T, tar(t) = n \Longrightarrow c(t_1, \ldots, t_n) \in \mathcal{T}^*$

Vordefinierte Typkonstanten: $\Rightarrow$ , $bool$
mit Arität $tar(\Rightarrow) = 2$, $tar(bool) = 0$.

# Signatur und Terme

Def (Signatur): $\Sigma = \langle \mathcal{T}, \Omega, ar \rangle$ wobei $\mathcal{T}$ Typdeklaration, $\Omega$ Menge von Operationen mit $ar : \Omega \to \mathcal{T}^*$

Def: Gegeben Signatur $\Sigma$, Menge $X$ von Variablen mit $type(x) \to \mathcal{T}^*$. Terme $\{T_{\Sigma,t}(X)\}_{t \in \mathcal{T}^*}$ sind kleinste Menge so dass

(i) $c \in \Sigma \Longrightarrow c \in T_{\Sigma,ar(c)}(X)$

(ii) $x \in X \Longrightarrow x \in T_{\Sigma,type(x)}(X)$

(iii) $r \in T_{\Sigma,t}(X), s \in T_{\Sigma,t \Rightarrow u}(X) \Longrightarrow r\, s \in T_{\Sigma,u}(X)$

(iv) $x \in X, s \in T_{\Sigma,type(x) \Rightarrow t}(X) \Longrightarrow \lambda\, x^{type(x)}.s \in T_{\Sigma,t}(X)$

Vordefinierte Operationen: siehe HOL.

# Spezifikationen und Theoreme

Def (Spezifikation): $\mathcal{S}p = \langle \Sigma, \mathcal{A}x \rangle$ mit Signatur $\Sigma$ und Menge von Axiomen (Terme $t \in T_{\Sigma,bool}(X)$

- Isabelle: Spezifikation $\equiv$ Theorie

Def (Theorem): $\phi \in T_{\Sigma,bool}(X)$, so dass $\mathcal{S}p \vdash \phi$ ($\phi$ in $\mathcal{S}p$ herleitbar)

# Spezifikationen und Theoreme

Def (Spezifikation): $\mathcal{S}p = \langle \Sigma, \mathcal{A}x \rangle$ mit Signatur $\Sigma$ und Menge von Axiomen (Terme $t \in T_{\Sigma,bool}(X)$

- Isabelle: Spezifikation $\equiv$ Theorie

Def (Theorem): $\phi \in T_{\Sigma,bool}(X)$, so dass $\mathcal{S}p \vdash \phi$ ($\phi$ in $\mathcal{S}p$ herleitbar)

- Herleitbarkeit: logischer Kalkül, z.B. natürliches Schließen in HOL
- Gültigkeit ($\mathcal{S}p \models \phi$): $\phi$ gilt in allen Modellen

# Spezifikationen und Theoreme

Def (Spezifikation): $\mathcal{S}p = \langle \Sigma, \mathcal{A}x \rangle$ mit Signatur $\Sigma$ und Menge von Axiomen (Terme $t \in T_{\Sigma,bool}(X)$

- Isabelle: Spezifikation $\equiv$ Theorie

Def (Theorem): $\phi \in T_{\Sigma,bool}(X)$, so dass $\mathcal{S}p \vdash \phi$ ($\phi$ in $\mathcal{S}p$ herleitbar)

- Herleitbarkeit: logischer Kalkül, z.B. natürliches Schließen in HOL
- Gültigkeit ($\mathcal{S}p \models \phi$): $\phi$ gilt in allen Modellen
- Korrektheit $\mathcal{S}p \vdash \phi \Longrightarrow \mathcal{S}p \models \phi$, Vollständigkeit $\mathcal{S}p \models \phi \Longrightarrow \mathcal{S}p \vdash \phi$
  - Eigenschaften des logischen Kalküls
  - HOL: nicht vollständig
  - FOL, Gleichungslogik: vollständig
  - Gödel: natürliche Zahlen + Konsistenz $\Longrightarrow$ Unvollständigkeit

# Implementation

- Implementation: Axiome müssen bewiesen werden.
- Beispiel: Implementation von Stack durch Listen
- Übersetzung von Typen und Operationen:
  - Abbildung der Typen: $stack \mapsto list$
  - Abbildung der Operationen $empty \mapsto Nil$, $push \mapsto Cons, \ldots$
  - Abbildung muss Aritäten bewahren.
  - Damit Übersetzung von Termen möglich.

# Signaturmorphismus

Def: Gegeben $\Sigma = \langle \mathcal{T}, \Omega \rangle, \Sigma' = \langle \mathcal{T}', \Omega' \rangle$. Ein Signaturmorphismus $\sigma : \Sigma \to \Sigma'$ ist $\sigma_T : T \to T', \sigma_\Omega : \Omega \to \Omega'$ so dass

(i) $tar(\sigma_T(t)) = tar(t)$, $\sigma_\Omega(\Rightarrow) = \Rightarrow$

Dann $\sigma_T^* : \mathcal{T}^* \to \mathcal{T}'^*$

$$\begin{aligned} \sigma_T^*(\alpha) &\stackrel{def}{=} \alpha \\ \sigma_T^*(c(t_1, \ldots, t_n)) &\stackrel{def}{=} \sigma_T(c)(\sigma_T^*(t_1), \ldots, \sigma_T^*(t_n)) \end{aligned}$$

(ii) $ar(\sigma_\Omega(c)) = \sigma_T^*(ar((c)))$

Dann $\sigma^* : T_\Sigma(X) \to T_{\Sigma'}(X)$

$$\begin{aligned} \sigma^*(x) &\stackrel{def}{=} x \\ \sigma^*(c) &\stackrel{def}{=} \sigma_\Omega(c) \\ \sigma^*(s\,t) &\stackrel{def}{=} \sigma^*(s)\,\sigma^*(t) \\ \sigma^*(\lambda\, x.t) &\stackrel{def}{=} \lambda\, x.\sigma^*(t) \end{aligned}$$

# Signaturmorphismus

Def: Gegeben $\Sigma = \langle \mathcal{T}, \Omega \rangle, \Sigma' = \langle \mathcal{T}', \Omega' \rangle$. Ein Signaturmorphismus $\sigma : \Sigma \to \Sigma'$ ist $\sigma_T : T \to T', \sigma_\Omega : \Omega \to \Omega'$ so dass

(i) $tar(\sigma_T(t)) = tar(t)$, $\sigma_\Omega(\Rightarrow) = \Rightarrow$

Dann $\sigma_T^* : \mathcal{T}^* \to \mathcal{T}'^*$

$$\begin{aligned} \sigma_T^*(\alpha) &\stackrel{def}{=} \alpha \\ \sigma_T^*(c(t_1, \ldots, t_n)) &\stackrel{def}{=} \sigma_T(c)(\sigma_T^*(t_1), \ldots, \sigma_T^*(t_n)) \end{aligned}$$

(ii) $ar(\sigma_\Omega(c)) = \sigma_T^*(ar((c)))$

Dann $\sigma^* : T_\Sigma(X) \to T_{\Sigma'}(X)$

$$\begin{aligned} \sigma^*(x) &\stackrel{def}{=} x \\ \sigma^*(c) &\stackrel{def}{=} \sigma_\Omega(c) \\ \sigma^*(s\,t) &\stackrel{def}{=} \sigma^*(s)\,\sigma^*(t) \\ \sigma^*(\lambda\, x.t) &\stackrel{def}{=} \lambda\, x.\sigma^*(t) \end{aligned}$$

Lemma: Übersetzung wohldefiniert $s \in T_{\Sigma,t}(X) \Longrightarrow \sigma^*(s) \in T_{\Sigma,\sigma_T^*(t)}(X)$

# Spezifikationsmorphismus

Def: Gegeben $\mathcal{S}p = \langle \Sigma, \mathcal{A}x \rangle, \mathcal{S}p' = \langle \Sigma', \mathcal{A}x' \rangle$ Ein Spezifikationsmorphismus (Theoriemorphismus) ist ein Signaturmorphismus $\sigma_\Sigma : \Sigma \to \Sigma'$ der Axiome, so dass $\phi \in \mathcal{A}x$, $\sigma^*(\phi) \in Thm(\mathcal{S}p')$.

Def: Eine Implementation für $\mathcal{S}p = \langle \Sigma, \mathcal{A}x \rangle$ ist eine Theorie $\mathcal{T}h = \langle \Sigma', \mathcal{A}x' \rangle$ und ein Spezifikationsmorphismus $\sigma : \mathcal{S}p \to \mathcal{T}h$.

- Signatur- und Theoriemorphismen strukturieren die Entwicklung

# Der Entwicklungszyklus

- Anforderungsspezifikation (requirement specification)
  - Axiomatisch
  - Unvollständig
- Entwurfsspezifikation (design specification)
  - Konsistent (konservativ, in HOL)
  - Vollständig
- Ausführbare Spezifikation (executable specification)
  - Ausführbar (funktionale Untermenge von HOL), testbar
  - Strukturiert

# Warteschlangen

- Eine Sorte, fünf Operationen (vgl. Stack)

```
typedecl 'a q
consts
  mtQ     :: "'a q"
  isEmpty :: "'a q => bool"
  enq     :: "'a => 'a q => 'a q"
  front   :: "'a q => 'a"
  deq     :: "'a q => 'a q"
```

# Warteschlangen

- Wann ist eine Schlange leer:

```
axioms
  empty: "isEmpty mtQ"
  notEmpty: "~ (isEmpty (enq a q))"
```

- Verhalten von `front`:

```
  front: "front (enq a q) = (if isEmpty q then a
                             else front q)"
```

- Verhalten von `deq`:

```
  deq:  "deq (enq a q) = (if isEmpty q then mtQ
                          else enq a (deq q))"
```

# Warteschlangen — Entwurfsspezfikation

- Eine Warteschlange hat einen Inhalt
- Selbe Signatur wie oben, plus:

```
consts
  cont     :: "'a q => 'a list"
axioms
  contEq:    "cont q= cont p ==> q= p"
```

- Spezifikation des Verhaltens über den Inhalt

# Warteschlangen — Entwurfsspezfikation

- Axiome oben ersetzt durch

```
axioms
  mt:         "cont (mtQ) = []"
  isMtCont:   "isEmpty q = (cont q = [])"
  enqCont:    "cont (enq a q) = cont q @ [a]"
  frontCont: "front q = hd (cont q)"
  deqCont:    "cont (deq q) =  tl (cont q)"
```

- Zusätzliches Axiom

```
  contEq:     "cont q= cont p ==> q= p"
```

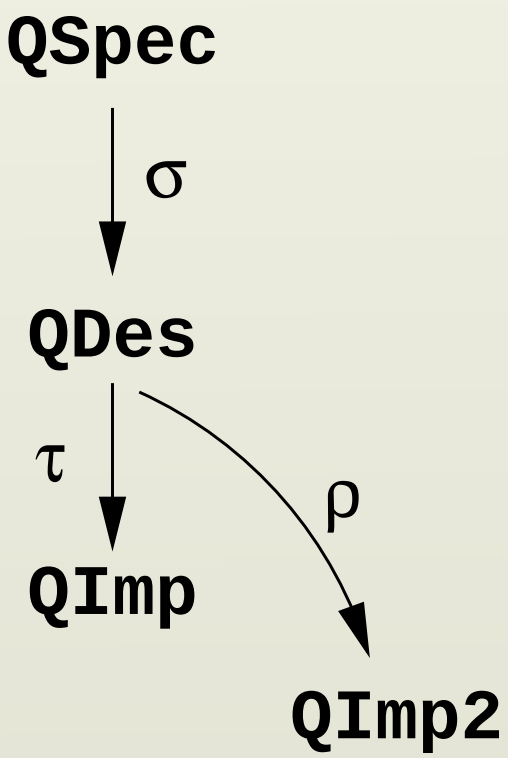
  - Vorherige Axiome können bewiesen werden

# Warteschlangen — Implementation

- Warteschlangen implementiert durch Listen

```
datatype 'a q = Q "'a list"
  ...
defs
  mt_def: "mtQ == Q []"
primrec
 "isEmpty (Q q)= null q"
primrec
 "enq x (Q q)= Q (x#q)"
primrec
 "front (Q q)= hd (rev q)"
primrec
 "deq (Q q)= Q (tl (rev q))"
```

# Warteschlangen — Struktur der Spezifikationen




- `QSpec` — Anforderungsspezifikation
- `QDes` — Entwurfsspezifikation
- `QImp` — Implementation
- `QImp2` — effizientere Implementation:

```
datatype 'a q =
  Q "'a list" "'a list"
```

Spezifikationsmorphismen können komponiert werden:
`QImp`, `QImp2` Implementationen von `QSpec`

# Abschließendes Beispiel

- Datenbank mit eindeutigen, generierten Schlüsseln:

```
consts
 init    :: db
 new_key :: db-> (db, key)
 lookup  :: db -> key -> data option
 upd     :: db -> key -> data-> db
 rem     :: db -> key -> db
```

  - Alternative:

```
lookup   :: db -> key -> data
contains :: db-> key-> bool
```

  - `...` aber mit `option` kürzer, Axiome lesbarer

# Abschließendes Beispiel

- Axiome: Verhalten von `lookup`:

```
axioms
 new_key db = (db', k) ==> lookup db' k = None
 lookup init k = Nothing
 lookup (upd (db k d)) k = Just d
 k ~= l ==> lookup (upd (db k d) l)= lookup db l
 lookup (rem db k) k = Nothing
 k ~= l ==> lookup (rem db k) l= lookup db l
```

# Abschließendes Beispiel

- Verhalten von `rem` und `update` (benötigt?)::

```
rem init = init
rem (upd db k d) k = rem db k
k ~= l ==>  rem (upd db k d) l = upd (rem db l) k d
upd (upd db k d) k d'= upd db k d'
k ~= l ==> upd (upd db k d) l d= upd (upd db l d) k d
```

- `new_key`: Seiteneffekt bei Generierung eines neuen Schlüssels
- Modellierung des Zustands explizit — kann umständlich werden
- Deshalb: imperative Modellierung $\rightsquigarrow$ Z (nächste Woche)

# Zusammenfassung

- Algebraische Spezifikationen:
  - Signatur deklariert Typen und Operationen
  - Axiome spezifieren Verhalten
- Signatur- und Theoriemorphismen
  - Implementation als Theoriemorphismus $\sigma : Spec \rightarrow Imp$
- Der Entwicklungszyklus:
  - Anforderungsspezifikation (axiomatisch)
  - Entwurfsspezifikation (konsistent)
  - Ausführbare Spezifikation (Prototyp)
- Nächste Woche: modellbasierte Spezifikation imperativer Programme

# Vorlesung vom 16.01.06:
# Z für Anfänger

# Wo sind wir?

- Teil I: Grundlagen formaler Logik
- Teil II: Modellierung
- Teil III: Spezifikationsformalismen
  - Axiomatische Spezifikation
  - Spezifikation mit Z: Grundlagen & Einführung
  - Z: Datentypen & Schemakalkül
  - Verfeinerung, Datenverfeinerung und Verhaltensgleichheit

# Was ist Z?

- Spezifikationssprache, basierend auf getypter Mengenlehre
  - Alles ist eine Menge (Mengen sind Typen)
  - Viel syntaktische Konvention
- Spezifikation von imperativen Programmen
  - Zustand und Zustandsänderung integraler Bestandteil
- Entwickelt Ende 80er, Oxford UCL und IBM UK

# Mengenlehre

- Begründet 1874–1884 durch Georg Cantor (Naive Mengenlehre)
  - Leider inkonsistent (Burali-Forte Paradox, Russelsches Paradox)
- Axiomatisierungen durch Zermelo (inkonsistent),
  später Fränkel (ZF), von Neumann, Gödel, Bernays
- Axiome:
  Extensionalität, Separation, Paarbildung, Vereinigung, Potenzmenge,
  Wohlfundiertheit, Ersetzung, Leere Menge, Unendlichkeit, Auswahl
  - Auswahlaxiom unabhängig vom Rest
- Getypte Mengenlehre: HOL
  - Mengen sind Prädikate sind Funktionen nach `bool`

# Meine erste Z-Spezifikation

Eine axiomatische Definition:

$$iroot : \mathbb{N} \rightarrow \mathbb{N}$$

---

$$\forall\, a : \mathbb{N} \bullet iroot(a) * iroot(a) \leq a < (iroot(a) + 1) * (iroot(a) + 1)$$

- Boxen kennzeichnen Spezifikationen
- Deklaration, dann Spezifikation (Paragraphen)

# Elemente von Z

- Sprache im kleinen:
  - Deklarationen
  - Ausdrücke
  - Prädikate
- Basiselemente: das mathematical toolkit
- Sprache im großen:
  - Schema
  - Schemakalkül

# Der Texteditor

- Beispiel: Editor zum Bearbeiten von Texten
- Typdeklaration und Typsynomym:

$$[CHAR]$$
$$TEXT == seq\ CHAR$$

  - $seq$: Listen in Z
  - Beliebige Abkürzungen mit $==$

# Der Texteditor

- Beispiel: Editor zum Bearbeiten von Texten
- Typdeklaration und Typsynomym:

$$[CHAR]$$
$$TEXT == seq\ CHAR$$

  - $seq$: Listen in Z
  - Beliebige Abkürzungen mit $==$
- Maximale Anzahl Zeichen im Editor:

$$maxsize : \mathbb{N}$$
$$\hline$$
$$maxsize \leq 65535$$

# Zustandsschemata

- Repräsentation von Zuständen in Z: Schema
- Editor hat internen Zustand (edierter Text):

$$\begin{array}{|l} \textit{Editor} \\ \hline left, right : TEXT \\ \hline \#(left \frown right) \leq maxsize \\ \hline \end{array}$$

- Schema haben Namen
- Schema sind geschlossen
- Prädikate sind Invarianten

# Anfangszustand

- Spezieller Name $Init$

$$\begin{array}{|l}
\hline
Init \\
Editor \\
\hline
left = right = \langle\,\rangle \\
\hline
\end{array}$$

- Benutzt Schema $Editor$
  - Schemas als Makros

# Zustandsübergang: Zeichen einfügen

- Axiomatische Definition: druckende Zeichen

$$printing : \mathbb{P}\, CHAR$$

- Zustandsübergang des Editors: Zeichen einfügen

*Insert*

$\Delta Editor$

$ch? : CHAR$

---

$ch? \in printing$

$left' = left \frown \langle ch? \rangle$

$right' = right$

- $\Delta$ deklariert Zustandsübergang
- Vorher: *left*, nachher: *left'*
- *ch?* Eingabevariable

# Cursorbewegung

$$right\_arrow : CHAR$$

---

$$right\_arrow \notin printing$$

- $right\_arrow$ nicht einfügen

*Forward*

$$\Delta Editor$$
$$ch? : CHAR$$

---

$$ch? = right\_arrow$$
$$left' = left \frown \langle head(right) \rangle$$
$$right' = tail(right)$$

# Cursorbewegung

$$right\_arrow : CHAR$$

---

$$right\_arrow \notin printing$$

- $right\_arrow$ nicht einfügen

*Forward*

$$\Delta Editor$$
$$ch? : CHAR$$

---

$$ch? = right\_arrow$$
$$left' = left \frown \langle head(right) \rangle$$
$$right' = tail(right)$$
$$right \neq \langle\,\rangle$$

- Undefiniert wenn $right = \langle\,\rangle$
- Implizite Vorbedingung explizit machen
- Operation partiell

# Cursorbewegung an Dateiende

Cursor an Dateiende:

$$EOF$$
$$Editor$$
$$right = \langle\,\rangle$$

Eingabe ist Pfeil nach rechts:

$$RightArrow$$
$$ch? : CHAR$$
$$ch? = right\_arrow$$

- Alles zusammen:

$$T\_Forward \mathrel{\widehat=} Forward \lor (EOF \land RightArrow \land \Xi Editor)$$

  - $\Xi S$: Zustand unverändert

$$\Xi Editor == \Delta Editor \land left' = left \land right' = right$$

# Mengen und Typen

- Mengen: Aufzählungen $\{red, green, blue\}$ oder $l \,..\, k$;
  vordefiniert: $\mathbb{N}, \mathbb{N}_1, \mathbb{Z}, \mathbb{R}$
- Mengen sind getypt: $\{1, 2, red\}$ Typfehler!
- Definition neuer Typen:
  - Freie Typen:

$$COLOUR ::= red \mid green \mid blue$$
$$NAT ::= null \mid cons\langle\langle NAT \rangle\rangle$$

  - Deklarationen: $[NAME]$
- Konstruktionen auf Mengen: $\cup, \cap, \setminus, \mathbb{P}, \mathbb{F}$

# Strukturierte Typen

- Kartesisches Produkt:

$$DATE == DAY \times MONTH \times YEAR \qquad heute = (16, 1, 2006)$$

- Binäre Relationen: $A \leftrightarrow B == \mathbb{P}(A \times B)$
  mit $\mathrm{dom}(A \leftrightarrow B) = \{a : A \mid \exists\, b \in B \bullet (a \mapsto b) \in A \leftrightarrow B\}$
  $\mathrm{ran}(A \leftrightarrow B) = \{b : B \mid \exists\, a \in A \bullet (a \mapsto b) \in A \leftrightarrow B\}$
- Partielle und totale Funktionen:

$$A \nrightarrow B == \{f : A \leftrightarrow B \mid \forall\, a : A, b_1, b_2 : B \bullet (a \mapsto b_1) \in f \land (a \mapsto b_2) \in f \Rightarrow b_1 = b_2\}$$
$$A \rightarrow B == \{f : A \nrightarrow B \mid \mathrm{dom}\, f = A\}$$

# Ausdrücke

- Ausdrücke:
  - Mengenlehre ($\cup$, $\cap$, Komprehension . . . )
  - Arithmetische und logische Operatoren
  - Kein vordefinierter Typ Bool
- Sequenzen:

$$\mathsf{seq}\, X == \{f : \mathbb{N} \pfun X \mid \mathsf{dom}\, f = 1 \,..\, \#f\}$$

  - Vordefinierte Operatoren: $head$, $tail$, $front$, $last$, $\frown$, in

# Der Geburtstagskalender

- Ziel: Kalender, der an Geburtstage erinnert

$$[NAME, DATE]$$

- Datenstrukturen

$$\begin{array}{|l} \textit{BirthdayBook} \\ \hline known : \mathbb{P}\, NAME \\ birthday : NAME \pfun DATE \\ \hline known = \mathrm{dom}\, birthday \end{array}$$

- Operationen: hinzufügen, finden, erinnern

# Geburtstag hinzufügen

$AddBirthday$

$\Delta BirthdayBook$

$name? : NAME$

$date? : DATE$

---

$name? \notin known$

$birthday' = birthday \cup \{name? \mapsto date?\}$

# Geburtstag finden

$FindBirtday$

$\Xi BirthdayBook$

$name? : NAME$

$date! : DATE$

---

$name? \in known$

$date! = birthday(name?)$

# An Geburtstage erinnern

*Remind*

$$\Xi BirthdayBook$$
$$today? : DATE$$
$$cards! : \mathbb{P}\, NAME$$

---

$$cards! = \{n \in known \mid birthday(n) = today\}$$

# Zusammenfassung

- Z: formale Spezifikationssprache
- Basiert auf getypter Mengenlehre
- Typen = Mengen = Prädikate
- Elemente von Z:
  - Deklarationen
  - Ausdrücke
  - Prädikate
- Vordefinierte Deklarationen, Operatoren im mathematical toolkit
- Schema beschreiben Zustand und Zustandsübergänge
  - Rechnen mit Schemata: Schemakalkül $\rightsquigarrow$ nächste Woche

# Vorlesung vom 23.01.06:
# Z für Fortgeschrittene

# Wo sind wir?

- Teil I: Grundlagen formaler Logik
- Teil II: Modellierung
- Teil III: Spezifikationsformalismen
  - Axiomatische Spezifikation
  - Spezifikation mit Z: Grundlagen & Einführung
  - Z: Schemakalkül & Verfeinerung
  - Datenverfeinerung und Verhaltensgleichheit

# Schema

- Schema sind Kurznotation
- Schreibweise:


$$Name \mathrel{\widehat=} [\, Declaration \mid Predicate \,]$$

  - Bindet $Name$ an das Schema
  - Schema haben eigenen Namensraum
  - Abkürzende Schreibweise (keine Semantik)

# Schema als Typen

- Bekannte Typen: Deklaration, freier Typ, Potenzmenge, Tupel
- Schematyp: benannte Tupel (binding)

$$
\begin{array}{|l}
\hline
SchemaOne \\
a : \mathbb{Z} \\
b : \mathbb{P}\,\mathbb{Z} \\
\hline
\end{array}
\qquad
\langle\!|\ a : \mathbb{Z}, b : \mathbb{P}\,\mathbb{Z}\ |\!\rangle
$$

  - Vgl. Tupel $\langle\!|\ a : \mathbb{Z}, b : \mathbb{P}\,\mathbb{Z}\ |\!\rangle \cong \mathbb{Z} \times \mathbb{P}\,\mathbb{Z}$
  - Reihenfolge irrelevant: $\langle\!|\ a : \mathbb{Z}, b : \mathbb{P}\,\mathbb{Z}\ |\!\rangle = \langle\!|\ b : \mathbb{P}\,\mathbb{Z}, a : \mathbb{Z}\ |\!\rangle$
  - Elemente: $\langle\!|\ a \rightsquigarrow 2, b :\rightsquigarrow \{3, 4\}\ |\!\rangle \in SchemaOne$
  - Selektion mit ., z.B. $S.a = 2$

# Schema als Deklarationen

- Jedes Schemata ist eine Deklaration

$$\{SchemaOne \mid a \in b\} \qquad \{a : \mathbb{Z}, b : \mathbb{P}\,\mathbb{Z} \mid a \in b\}$$

- Charakterisierende Bindung (characteristic binding) $\theta S$

$$\theta SchemaOne = \langle\!| \; a \rightsquigarrow a, b \rightsquigarrow b \; |\!\rangle$$

  - Nur definiert in Kontext, in dem $a$ und $b$ deklariert sind

# Schema als Prädikate

- Jedes Schema ist ein Prädikat:
  - Allquantifiziert über Deklarationen,
  - Konjunktion der Prädikate

$SchemaTwo$

$SchemaOne$

---

$a \in b$

$a \geq 10$

$$\forall\, a : \mathbb{Z}, b : \mathbb{P}\,\mathbb{Z} \bullet a \in b \land a \geq 10$$

- Normalisierung von Schema: alle Einschränkungen in Prädikaten

# Umbenennung

- Komponenten können umbenannt werden: $Schema[new/old]$

$$OtherSchema \mathrel{\widehat{=}} SchemaTwo[x/a, y/b]$$

$OtherSchema$

$x : \mathbb{Z}$

$y : \mathbb{P}\,\mathbb{Z}$

---

$x \in y \land x \geq 10$

# Generische Schema

- Parametrisierte Schemata:

$SchemaThree\,[X]$

$a : X$

$b : \mathbb{P}\, X$

---

$a \in b$

- $X$ steht für beliebigen Typ
- Instantiierung: $SchemaThree[\mathbb{Z}]$

# Generische Schema

- Parametrisierte Schemata:

$SchemaThree\,[X]$

$a : X$

$b : \mathbb{P}\, X$

---

$a \in b$

- $X$ steht für beliebigen Typ
- Instantiierung: $SchemaThree[\mathbb{Z}]$

- Analog: generische Definitionen

$[X, Y]$

$$first : X \times Y \to X$$

---

$$\forall X : X, y : Y \bullet first(x, y) = x$$

$[X]$

$$\_ \frown \_ : \mathsf{seq}\ X \times \mathsf{seq}\ X \to \mathsf{seq}\ X$$

$$rev : \mathsf{seq}\ X \to \mathsf{seq}\ X$$

---

$$\forall s, t : \mathsf{seq}\ X \bullet s \frown t = s \cup \{n : \mathsf{dom}\ t \bullet n + \#s \mapsto t(n)\}$$

$$\forall s : \mathsf{seq}\ X \bullet revs = (\lambda\, n : \mathsf{dom}\ s \bullet s(\#s - n + 1))$$

# Der Schemakalkül

Die Operationen im Überblick:

- Konjunktion, Disjunktion und Negation
- Quantifikation und Restriktion
- Dekoration, $\Delta$ und $\Xi$
- Komposition
- Pipes

# Konjunktion und Disjunktion

- $S \wedge T$: Vereinigung der Deklaration, Konjunktion der Prädikate
  - In beiden Schema deklarierte Variablen: gleicher Typ
- $S \vee T$ : Vereinigung der Deklaration, Disjunktion der Prädikate
  - In beiden Schema deklarierte Variablen: gleicher Typ
- $\neg S$ : Deklaration unverändert, Negation der Prädikate in toto
  - Meist nicht so sinnvoll

# Quantifizierung und Restriktion

- $\forall\, x_1 : T_1 \ldots x_n : T_n \bullet [\, Decl \mid P \,]$:
  - Variablen $x_1, \ldots x_n$ aus $Decl$ entfernen
  - $P$ ersetzen durch $\forall\, x_1 : T_1, \ldots, x_n : T_n \bullet P$
  - Deklaration in $Decl$: gleicher Typ
- $\exists\, x_1 : T_1 \ldots x_n : T_n \bullet [\, Decl \mid P \,]$: analog
  - Erlaubt verdecken von Bezeichnern
- Abkürzende Notation: $S \setminus L$

$$SchemaTwo \setminus (a) = [\, b : \mathbb{P}\,\mathbb{Z} \mid \exists\, a : \mathbb{Z} \bullet a \in b \land a \geq 10 \,]$$

# Dekoration, $\Delta$ und $\Xi$

- $S'$: wie $S$, aber alle Deklarationen und Prädikate mit $'$
- $S?, S!$: analog
- Weitere Abkürzungen:

$$\Delta Schema \mathrel{\widehat=} [\, Schema;\ Schema' \,]$$

$$\Xi Schema \mathrel{\widehat=} [\, \Delta Schema \mid \theta Schema = \theta Schema' \,]$$

- $\Xi$ kann überschrieben werden

# Komposition

- Schema = Zustandsübergang = Operation, dann Komposition von Operationen
- Voraussetzung: Für jede gestrichene Deklaration in Quelle ungestrichene Deklaration in Ziel

$$S \fatsemi T \mathrel{\widehat=} \exists\, State'' \bullet\ (\exists\, State' \bullet [\, S;\ State' \mid \theta State' = \theta State'' \,]) \land (\exists\, State \bullet [\, T;\ State \mid \theta State = \theta State'' \,])$$

# Pipes

- Pipe: Verknüpfung der Ausgabe mit der Eingabe
- Voraussetzung:
  Für jede Ausgabevariable in Quelle Eingabevariable in Ziel

$$S \gg T \mathrel{\widehat{=}} \exists\, Pipe!? \bullet \; (\exists\, Pipe! \bullet [\, S;\; Pipe!? \mid \theta Pipe! = \theta Pipe!?) \land (\exists\, Pipe? \bullet [\, T;\; Pipe!? \mid \theta Pipe? = \theta Pipe!?)$$

# Zusammenfassung

- Schema: abkürzende Schreibweise für . . .
  - Typen
  - Deklarationen
  - Prädikate
- Schema können Systemzustände modellieren
- Schemakalkül:
  - Log. Operatoren
  - Zustandsübergänge, $\Delta$, $\Xi$
  - Komposition
- Nächste Woche: Programmentwicklung

# Vorlesung vom 30.01.06: Techniken der formalen Programmentwicklung

# Wo sind wir?

- Teil I: Grundlagen formaler Logik
- Teil II: Modellierung
- Teil III: Spezifikationsformalismen
  - Axiomatische Spezifikation
  - Spezifikation mit Z: Grundlagen & Einführung
  - Z: Schemakalkül
  - Formale Programmentwicklung: Endrekursion & Datenverfeinerung

# Formale Programmentwicklung

- Von der Spezifikation zum Code:

$$SP_1 \rightsquigarrow SP_2 \rightsquigarrow \ldots \rightsquigarrow SP_n$$

- Einzelne Schritte:
  - Theoriemorphismen: Modellverfeinerung

# Formale Programmentwicklung

- Von der Spezifikation zum Code:

$$SP_1 \rightsquigarrow SP_2 \rightsquigarrow \ldots \rightsquigarrow SP_n$$

- Einzelne Schritte:
  - Theoriemorphismen: Modellverfeinerung
  - Funktionen implementieren:

    Rekursion $\longrightarrow$ Endrekursion $\longrightarrow$ Iteration

# Formale Programmentwicklung

- Von der Spezifikation zum Code:

$$SP_1 \rightsquigarrow SP_2 \rightsquigarrow \ldots \rightsquigarrow SP_n$$

- Einzelne Schritte:
  - Theoriemorphismen: Modellverfeinerung
  - Funktionen implementieren:
    Rekursion $\longrightarrow$ Endrekursion $\longrightarrow$ Iteration
  - Datenverfeinerung: Datentypen implementieren

# Endrekursion

- Def: Eine Funktion $f : S \Rightarrow T$

$$f\ x \equiv if\ B\,x\ then\ f(K\,x)\ else\ H\,x$$

ist endrekursiv und entspricht der Iteration

```
[[T]] f([[S]] x)
{
  [[S]] a= x;
  while B(a) { a= K(a); }
  return H(a);
}
```

- [[T]]: Übersetzung des Typen $T$

# Kriterien für Endrekursivität

- Genau ein rekursiver Aufruf (lineare Rekursion)
- Genau eine Fallunterscheidung
- Rekursiver Aufruf außen (unter Fallunterscheidung)

# Beispiele

- Berechung des Modulus:

```
consts mod :: "nat* nat=> nat"
recdef mod  "measure fst"
  "mod (x,y) = (if x >= y & y > 0 then mod (x-y ,y)
                                  else x)"
```

# Beispiele

- Berechung des Modulus:

```
consts mod :: "nat* nat=> nat"
recdef mod  "measure fst"
  "mod (x,y) = (if x >= y & y > 0 then mod (x-y ,y)
                                  else x)"
```

- Berechnung des ggT:

```
consts gcd :: "nat * nat => nat"
recdef gcd  "measure (%(x, y). x + y)"
  "gcd (x, 0) = x"
  "gcd (x, y) = gcd (y, mod(x, y))"
```

# Von der Rekursion zur Endrekursion

- Gegenbeispiel — nicht endrekursiv:

```
consts fact :: "nat => nat"
recdef fact  less_than
  "fact x = (if x = 0 then 1
                      else x * fact (x - 1))"
```

# Von der Rekursion zur Endrekursion

- Gegenbeispiel — nicht endrekursiv:

```
consts fact :: "nat => nat"
recdef fact  less_than
  "fact x = (if x = 0 then 1
                      else x * fact (x - 1))"
```

- Wie von Rekursion zur Endrekursion?
- Endergebniss in weiterem Parameter akkumulieren.

# Von der Rekursion zur Endrekursion

- Dazu Hilfsfunktion

```
constdefs fact :: "nat => nat"
  "fact x == fact'(x, 1)"

consts fact' :: "(nat, nat) => nat"
recdef fact'  "measure fst"
  "fact' x a = (if x = 0 then a
                         else fact' (x - 1, x*a)"
```

- Technik läßt sich verallgemeinern

# Überführung in Endrekursion

- Voraussetzung: lineare Rekursion
  - d.h. genau ein rekursiver Aufruf
- Gegeben

$$\begin{array}{lcl} f & :: & S \Rightarrow T \\ f(x) & = & if\, B(x)\ then\, \phi(f(K(x)), E(x))\ else\, H(x) \end{array}$$

  mit $K : S \Rightarrow S, \phi : T \times R \Rightarrow T, E : S \Rightarrow R, H : S \Rightarrow T$
- Überführung in endrekursive Form durch Hilfsfunktion

# Einführung von Endrekursion

- Aufrufschema: nach $n$ Aufrufen

$$f(x) = \phi(\phi(\cdots(\phi(b, a_{n-1}), \ldots), a_1), a_0)$$
$$a_i = E(K^i(x)), b = H(K^n(x))$$

# Einführung von Endrekursion

- Aufrufschema: nach $n$ Aufrufen

$$f(x) = \phi(\phi(\cdots(\phi(b, a_{n-1}), \ldots), a_1), a_0)$$
$$a_i = E(K^i(x)), b = H(K^n(x))$$

- Voraussetzung: $\phi$ assoziativ, $\phi(\phi(r, s), t) = \phi(r, \phi(s, t))$
- Dann gilt

$$\begin{aligned} f(x) &= \phi(\phi(\phi(\cdots(\phi(b, a_{n-1}), \ldots), a_2), a_1), a_0) \\ &= \phi(\phi(\cdots(\phi(b, a_{n-1}), \ldots), a_2), \phi(a_1, a_0)) \\ &= \phi(\cdots(\phi(b, a_{n-1}), \ldots), \phi(a_2, \phi(a_1, a_0))) \\ &= \phi(b, \phi(a_{n-1}, \ldots \phi(a_2, \phi(a_2, \phi(a_1, a_0))))) \end{aligned}$$

- Das ist endrekursiv!

# Transformationsregel

- Für $f : S \Rightarrow T$ mit

$$f(x) = if\, B(x)\ then\, \phi(f(K(x)), E(x))\ else\, H(x)$$

mit $K : S \Rightarrow S, \phi : T \times T \Rightarrow T, E : S \Rightarrow T, H : S \Rightarrow T,$

- Wenn $\phi$ assoziativ und $e$ neutrales Element:

$$\phi(\phi(r, s), t) = \phi(r, \phi(s, t)) \qquad \phi(r, e) = r$$

- Dann ist die äquivalente endrekursive Formulierung:

$$\begin{aligned} f(x) &= g(x, e) \\ g(x, y) &= if\, B(x)\ then\, g(K(x), \phi(E(x), y))\ else\, \phi(H(x), y) \end{aligned}$$

# Beispiel: Länge einer Liste

- Rekursive Definition:

```
consts length :: "'a list=> nat"
recdef length measure size
 length []     = 0
 length (x#xs) = 1+ length xs
```

- Endrekursiv:

```
constdefs length ::  "'a list=> nat"
 "length xs == length'(xs, 0)"
consts length' :: "'a list* nat=> nat"
recdef length' "measure fst"
  length' ([], n)   = n
  length' (x#xs, n) = length'(xs, n+1)"
```

# Beispiel: Listenreversion

- Rekursive Definition:

```
consts rev :: "'a list=> 'a list"
recdef rev "measure size"
  rev []     = []
  rev (x#xs) = (rev xs)@[x]
```

-

# Beispiel: Listenreversion

- Rekursive Definition:

```
consts rev :: "'a list=> 'a list"
recdef rev "measure size"
  rev []     = []
  rev (x#xs) = (rev xs)@[x]
```

- Endrekursiv:

```
constdefs rev ::  "'a list=> 'a list"
 "rev xs == rev'(xs, [])"
consts rev' :: "'a list=> 'a list=> 'a list"
recdef rev' "measure size"
  rev'([], ys)   = ys
  rev'(x#xs, ys) = rev'(xs, x#ys)
```

# Datenverfeinerung

- Problem: Abstrakten Datentyp (`List`, `Set`) implementieren
- Abbildung der Operationen
- Axiome gelten? Welche?
- Verfeinerung auf Datenebene (Reification)

# Datentypen (Z)

- Def (Datentyp): $\mathcal{X} = \langle X, xi, \{xop_\omega\}_{\omega \in \Sigma} \rangle$ mit
  - Globaler Zustandsraum $G$
  - Lokaler Zustandsraum $X$
  - Initialisierung $xi : G \leftrightarrow X$ (total)
  - Operationen $xop_\omega : X \times G \leftrightarrow X \times G$ (partiell)

# Datentypen (Z)

- Def (Datentyp): $\mathcal{X} = \langle X, xi, \{xop_\omega\}_{\omega \in \Sigma} \rangle$ mit
  - Globaler Zustandsraum $G$
  - Lokaler Zustandsraum $X$
  - Initialisierung $xi : G \leftrightarrow X$ (total)
  - Operationen $xop_\omega : X \times G \leftrightarrow X \times G$ (partiell)
- $\mathcal{C}$ verfeinert $\mathcal{A}$, wenn für alle Programme $P(\mathcal{X})$ gilt $P(\mathcal{C}) \subseteq P(\mathcal{A})$ (globaler Zustandsraum gleich)
- Idee: $\mathcal{C}$ ist determinierter und totaler als $\mathcal{A}$
- Problem: Wie beweisen?

# Simulationen

Def (Simulation): Sei $\mathcal{A} = \langle A, ai, \{aop_i\}_{i\in\Sigma}\rangle$ and $\mathcal{C} = \langle C, ci, \{cop_i\}_{i\in\Sigma}\rangle$.
$\mathcal{C}$ simuliert $\mathcal{A}$ wenn es $R : A \leftrightarrow C$ gibt so dass

$$
\begin{array}{ll}
(Init) & \forall\, c' \in C, g \in G \cdot (g, c') \in ci \\
& \Longrightarrow \exists\, a' \in A \cdot (g, a') \in ai \wedge (a', c') \in R \\
(Appl) & \forall\, i \in \Omega \forall\, a \in A, c \in C, g \in G \cdot \\
& \quad (a, g) \in \mathsf{dom}\ aop_i \wedge (a, c) \in R \\
& \quad \Longrightarrow (c, g) \in \mathsf{dom}\ cop_i \\
(Corr) & \forall\, i \in \Omega \forall\, a \in A, c, c' \in C, g, g' \in G \cdot \\
& \quad (a, g) \in \mathsf{dom}\ aop_i \wedge (a, c) \in R \wedge ((c, g), (c', g')) \in cop_i \\
& \quad \Longrightarrow \exists\, a' \in A \cdot ((a, g), (a', g')) \in aop_i \wedge (a', c') \in R
\end{array}
$$

Satz: $\mathcal{C}$ simuliert $\mathcal{A}$, dann $P(\mathcal{C}) \subseteq P(\mathcal{A})$

# Beispiel

- Stacks mit $\Sigma = \{push, pop, top\}$

  ```
  types globalstate= "int* int"
  ```

- Abstrakt: Stack als Liste
- Konkret: Stack als Feld mit Zeiger
- Problem: keine direkte Verfeinerung
  - Axiom a2 gilt nicht
  - Aber: Gleichheit auf lokalem Zustandsraum nicht sichtbar
  - Trotzdem sinnvolle Verfeinerung $\rightsquigarrow$ Datenverfeinerung

# Stacks durch Listen

```
types lstate = int list
constdefs
 lempty :: "(globalstate* lstate) set"
"lempty == {(g, l). l= []}"
 lpush :: "((globalstate* lstate)* (globalstate* lstate)) set"
"lpush == {(((i, out), l), ((i', out'), l')).
           (i', out')= (i, out) & (l'= i # l)}"
 lpop  :: "((globalstate* lstate)* (globalstate* lstate)) set"
"lpop  == {((g, l), (g', l')).
   l ~= [] & g'= g & l'= tl l}"
 ltop  :: "((globalstate* lstate)* (globalstate* lstate)) set"
"ltop  == {(((i, out), l), ((i', out'), l')).
           l ~= [] & i'= i & out'= hd l & l= l'}"
```

# Stacks durch Felder

```
types astate = "nat* (nat ~=> int)"
constdefs
 aempty :: "(globalstate* astate) set"
"aempty == {(g, l). l= (0, empty)}"
 apush :: "((globalstate* astate)* (globalstate* astate)) set"
"apush == {(((i, out), (p, a)), ((i', out'), (p', a'))).
           (i', out')= (i, out) & p'= p+ 1 & a'= a(p |-> i)}"
 apop  :: "((globalstate* astate)* (globalstate* astate)) set"
"apop  == {((g, (p, a)), (g', (p', a'))).
            g'= g & p'= p- 1 & a'= a}"
 atop  :: "((globalstate* astate)* (globalstate* astate)) set"
"atop  == {(((i, out), (p, a)), ((i', out'), (p', a'))).
            i'= i & out'= the (a (p - 1)) & p'= p & a'= a}"
```

# Verfeinerung

- Die Verfeinerungsrelation:

```
constdefs
  r :: "(lstate* astate) set"
 "r == {(l, (p, a)). p= length l &
                     (! i. 0<= i & i< p -->
                          l ! i = the (a (p-i- 1)))}"
```

- Damit die drei Bedingungen zeigen.

# Zusammenfassung

- Schrittweise Verfeinerung: von Spezifikation zum Programm
- Rekursion $\longrightarrow$ lineare Rekursion $\longrightarrow$ Endrekursion
- Transformationsregeln: Schema für Entwurfsschritte
- Eigenschaftserhaltende Wechsel von Datentypen:
  - Datenverfeinerung
  - Verhaltensgleichheit (observational equivalence)

# Vorlesung vom 06.02.06: Zusammenfassung, Abschließende Bemerkungen & Ausblick

# Zusammenfassung

- Teil I: Grundlagen
  - Formale Logik
- Teil II: Modellierung
  - Datentypen, imperative und funktionale Programme
- Teil III: Spezifikation
  - CASL, Z und formale Programmentwicklung

# Teil I: Grundlagen

- Formale Logik: Grundlage der Informatik
- Grundbegriffe:
  - Konsistenz, Vollständigkeit, Entscheidbarkeit
- Formales Schließen in Kalkülen
  - Natürliches Schließen, Sequenzenkalkül

# Formale Logiken: von einfach bis kompliziert

- Offene Aussagenlogik $A \wedge B$
  - Entscheidbar, vollständig
- Prädikatenlogik erster Stufe $\forall\, x.P(x)$
  - Unentscheidbar, vollständig
- Prädikatenlogik höherer Stufe $\forall\, P.P \longrightarrow P$
  - Unentscheidbar, unvollständig
- Gödel's Unvollständigkeitssatz
  - Natürliche Zahlen + Konsistenz = Unvollständigkeit

# Klassisch vs. konstruktiv

- Klassische Logik: tertium non datur
  - HOL: Klassisches Logik höherer Stufe, einfach getypt (Church)
- Konstruktivistische Logik & Curry-Howard-Isomorphismus
  - Finitärer Konstruktivismus
  - Konstruktivismus (Bishop, Dummett)
  - Beweis in konstruktiver Logik = Term (Program) im $\lambda$-Kalkül
- Intuitionistische Logik (Brouwer)

# Theorembeweiser

- `Isabelle`: klassische Logik höherer Stufe
- Gleiche/ähnliche Logik: `HOL`, `HOLlight`, `ProofPower`
- Typtheorien:
  - Abhängige Typen (dependent types, z.B.. $\Pi n : \mathbb{N}.Vec(n)$).
  - Calculus of constructions: `Coq`
  - Martin-Löf-Typtheorie: `Alf`
- `PVS`: eigene Logik (abhängige Typen, Subtypen, . . . )

# Teil II: Modellierung

- Modellierung von Datentypen
  - Isabelle: Konstruktion von Datentypen durch Repräsentation und Abstraktion
  - Algebraische Datentypen
- Modellierung von rekursiven Funktionen
  - In HOL: nur totale Funktionen; wohlfundierte Rekursion
  - Cpo's und LCF: alle rekursiven Funktionen, aber Stetigkeitsbeweise
- Modellierung von imperative Programme
  - Denotationelle, operationelle und axiomatische Semantik
  - Explizites Modell des Zustandsübergangs

# Teil III: Formale Softwareentwicklung

- Der Entwicklungszyklus
  - Anforderungsspezifikation: axiomatisch, unvollständig
  - Entwurfsspezifikation: konsistent, vollständig
  - Ausführbare Spezifikation
  - Implementation
- Prozessmodelle:
  - Wasserfall-Modell, V-Modell, Spiralmodell
  - Evolutionäre formale Softwareentwicklung

# Spezifikationsformalismen

- Algebraisch (CASL)
  - Basiert auf universeller Algebra
  - Axiomatisch
- Modellbasiert (Z)
  - Basiert auf getypter Mengenlehre
  - Schemakalkül, imperative Programme
- Weitere Sprachen und Werkzeuge
  - VDM; B; ASM
  - Model Checker — automatische Beweisewerkzeuge
  - Viele Werkzeuge und Sprachen für nebenläufige Systeme (CSP/FDR, SPIN)

# Formale Programmentwicklung

- Verfeinerung
  - Theoriemorphisms $\sigma : SP_1 \rightarrow SP_2$
  - Endrekursion
  - Transformationsregeln (z.B. Divide-and-conquer)
- Datenverfeinerung
  - Observational and behavioural equivalence: gleich ist nicht gleich
  - Beispiel: Warteschlangen

# Anwendung formaler Methoden in der Praxis

- Z: CICS, B: Meteor (führerlose Metro)
- Praxis High Integrity Software (UK): Z & SPARK
- Microsoft:
  - SLAM: Automatische Verifikation von Treibern (Kombination aus model checking und Abstraktion)
  - Spec#: Objektorientierte Spezifikation (Interaktives Beweisen); AsmL: Abstract state machines (model checking)
- Intel: Verifikation der Fließkommarithmetik für Pentium IA-32, IA-64
  - Kombination aus model checking (SDT, symbolic trajectory evaluation) und interaktivem Beweisen (Forte & FL)

# Formale Methoden in Bremen

- Deadlock- & Livelock-Analyse für die ISS
- Die CASL-Sprachfamilie
- Der autonome Rollstuhl Rolland
- Sicherheit für autonome mobile Systeme

# Die Zukunft

- . . . hat gerade erst angefangen
- Formale Methoden werden immer mehr Standard
- Projekt VeriSoft

# Nächstes Semester

- Kurs Formale Methoden der Softwareentwicklung II:
  - Forschungsnäher
  - Verifikation imperativer Programme, oder
  - Formale Programmentwicklung
- Seminar Formale Methoden der Softwareentwicklung

# Nächstes Semester

- Kurs Formale Methoden der Softwareentwicklung II:
  - Forschungsnäher
  - Verifikation imperativer Programme, oder
  - Formale Programmentwicklung
- Seminar Formale Methoden der Softwareentwicklung
- Darüber hinaus:
  - Diplomarbeiten
  - Studentische Hilfskräfte