6. Formale Beschreibung von Programmiersprachen

Die Beschreibung von Programmiersprachen gliedert sich in Syntax und Semantik. Allgemein wird gesagt, daß die **Syntax** das **textuelle Erscheinungsbild** des Programms definiert, während die **Semantik** seine **Bedeutung** festlegt. Der eigentliche Sachverhalt ist aber komplizierter: Auch die Wahl der Syntax wird von der erwünschten Bedeutung des Programms beeinflußt.

Fest steht, daß die Syntax die Menge von Textfolgen (Strings) definiert, die legale Programme in der gegebenen Programmiersprache darstellen. Die Legalität eines Programms wird von einer Spracherkennungskomponente, dem **Parser**, festgestellt. Nur legale Programme dürfen ausgeführt werden. Die Semantik beschreibt dann das Verhalten der legalen Programme. Die folgenden Ausdrücke sind in ML legal, d.h. syntaktisch korrekt:

(1) `1 div 0;`
(2) `let val y = 0 in 1 div y end;`
(3) `fun f x y = x div y; f a b;`

Man kann sich fragen, ob es sinnvoll ist, diese Ausdrücke in ML zuzulassen. Ausdruck (1) ist immer undefiniert, d.h. kann niemals erfolgreich evaluiert werden. Dasselbe gilt für Ausdruck (2), obwohl es dort schwerer zu ersehen ist. Nur in Ausdruck (3) besteht die Möglichkeit einer erfolgreichen Auswertung, nämlich, wenn `b` nicht Null ist.

Es sollte die Aufgabe einer Sprachdefinition sein, offensichtlich unsinnige Programme auszuschließen. Damit sollten Ausdruck (1) und (2) eigentlich nicht in ML zugelassen werden. Sie werden es aber dennoch, da entschieden wurde, daß ein Ausschluß die Syntaxbeschreibung der Sprache zu sehr komplizieren würde.

Im Prinzip können Spracheigenschaften, die **vor der Ausführung des Programms** entscheidbar sind, syntaktisch festgelegt werden. Nur was vor der Programm-ausführung unentscheidbar ist, z.B. weil es abhängig von Eingabedaten ist, muß semantisch behandelt werden. In der Praxis wird die Grenze von Syntax zu Semantik jedoch wesentlich konservativer angelegt, d.h. es werden wesentlich weniger Spracheigenschaften syntaktisch definiert, als prinzipiell möglich.

Fazit: Die Grenze zwischen Syntax und Semantik ist gleitend und wird weniger von

konzeptionellen als von rein technologischen Erwägungen bestimmt.

Formalität in der Beschreibung von Syntax und Semantik hat wesentliche Vorteile:

(1) Sie ermöglicht eine **Präzision**, die es erlaubt, die Beschreibung als **Referenz** für Benutzer und Implementierer zu verwenden.
(2) Sie ermöglicht eine leichtere und verläßlichere **Automation** der Programmerkennung, Programmentwicklung, Programmimplementierung und des Beweisens von Programmeigenschaften.
(3) Sie deckt Probleme und Widersprüche auf und hilft im Entwurf einer vernünftigen Sprache.

6.1 Syntax

Eine Sprache mit formal definierter Syntax heißt **formale Sprache.** Sie werden in der Vorlesung “Informatik IV” sehen, daß es verschiedene Klassen von formalen Sprachen gibt. Hier betrachten wir zunächst die in der Praxis wichtigste Klasse der **kontextfreien** Sprachen.

6.1.1 BNF–Notation

Diese Notation wurde maßgeblich von John Backus (dem Entwickler von FORTRAN) und Peter Naur für die Definition von Algol 60 entwickelt.
("BNF" steht für "Backus–Naur–Form".)

Definition: (Formale Sprache)
Sei $\sum$ eine Menge von Symbolen ("Alphabet"). Eine **formale Sprache** über $\sum$ ist eine Menge L von Worten aus $\sum^*$, d.h. $L \subseteq \sum^*$,wobei

$$\sum^* = \{\sigma_1...\sigma_n \mid n \geq 0, \sigma_i \in \sum\}$$

$\sigma_1...\sigma_n$ heißt **Wort** (man kann es auch als String verstehen). Der Iterationsoperator * heißt **Kleene–Star**. (S. C. Kleene ist ein berühmter amerikanischer Informatiker.)

Eine (kontextfreie) formale Sprache wird durch eine (BNF–) Grammatik beschrieben.

Definition: (BNF–Grammatik)

Eine **BNF–Grammatik** G ist ein Quadrupel

$$G = (N,\Sigma,P,s)$$

N ist eine Menge von **nicht–terminalen** Symbolen,

Σ ist eine Menge von **terminalen** Symbolen (das Alphabet), $N \cap \Sigma = \emptyset$,

P ist eine Menge von **Produktionen** der Form

$\alpha_i ::= \beta_{i1} \mid ... \mid \beta_{in}$,

wobei

für alle i,j gilt: $\alpha_i \in N$ und $\beta_{ij} \in (N \cup \Sigma)^*$,

s ist das Startsymbol, $s \in N$.

Bemerkungen:

– Der Begriff "kontextfrei" ist dadurch bedingt, daß α_i jeweils nur ein einziges Nichtterminal ist.

– Was rechts vom Produktionszeichen ::= steht, wird auch **BNF–Ausdruck** genannt.

– Die Sprache, die von Grammatik G erzeugt wird, wird mit L(G) bezeichnet.

– In der BNF–Notation werden Nichtterminale in spitzen Klammern eingerahmt (um sie von den Terminalen zu unterscheiden).

Die Bedeutung der BNF–Notation ist durch folgende Regeln festgelegt:

Sei $\sigma \in \Sigma$ und seien x und y BNF–Ausdrücke für die Sprachen X bzw. Y.

(1) Terminale Zeichen:
σ ist ein BNF–Audruck für die Sprache $\{\sigma\}$.

(2) Vereinigung:
x | y ist ein BNF–Ausdruck für die Sprache $X \cup Y$.

(3) Konkatenation:
x y ist ein BNF–Ausdruck für die Sprache $\{s_x \circ s_y \mid s_x \in X \wedge s_y \in Y\}$
($\circ$ ist Wortkonkatenation).

(4) Option, Iteration, Klammerung:
(a) {x} ist ein BNF–Ausdruck für die Sprache $X \cup \{\varepsilon\}$
(b) $\{x\}^*$ ist ein BNF–Ausdruck für die Sprache $\{s_1 \circ ... \circ s_n \mid n>0, s_1,...,s_n \in X\} \cup \{\varepsilon\}$
(c) $\{x\}^+$ ist ein BNF–Ausdruck für die Sprache $\{s_1 \circ ... \circ s_n \mid n>0, s_1,...,s_n \in X\}$
(d) [x] ist ein BNF–Ausdruck der Sprache X.

(ε bezeichnet das leere Wort).

Bemerkungen:

– (1)-(4) erweitert die Mächtigkeit der BNF–Notation nicht. Es können sogar alle erweiterten Grammatiken, deren Produktionen auf den rechten Seiten Vereinigung, Option, Iteration oder Klammerung enthalten auf Produktionen der Form

  $\alpha ::= \beta$ mit $\alpha \in N, \beta \in (N \cup \Sigma)*$

  zurückgeführt werden (siehe Übung).

– Sollen die Symbole {,},|,$^+$,* auch in Σ vorkommen, so müssen diese von denen der BNF–Notation unterschieden werden (z.B. durch Fettdrucken oder Unterstreichen).

Beispiele:

(1) Englische Satzbildung für Anfänger

Sei Σ = {the, who, man, woman, girl, cat, loves, meets, avoids},

N = {<Satz>, <NP>, <VP>, <N>, <V>}, s = <Satz>

<Satz> ::= <NP><VP>

<NP> ::= the <N> | the <N> who <VP>

<VP> ::= <V> | <V> <NP>

<N> ::= man | woman | girl | cat

<V> ::= loves | meets | avoids

(2) Sei Σ = {a,b}.

<one a in b> ::= $\{b\}^* a \{b\}^*$

<repeat ab> ::= $\{ab\}^+ = ab\{ab\}^*$

Das Produktionszeichen ::= ist dann einfach als Namensgebung für einen BNF–Ausdruck zu verstehen.

Bemerkung:

Für die Java-Syntax wird üblicherweise geschrieben

Nichtterminalsymbole	kursiv
Terminalsymbole	Courier

Jede Regel hat die Form

A: Ausdruck

(d.h. „:“ anstelle „:=“).

Für x | y schreibt man

x

y

(Zeilenumbruch).

Für die Option {x} schreibt man x_{opt}.

Bemerkung:

BNF–Grammatiken erlauben Rekursion in den Produktionen! Man muß aufpassen, daß die Bedeutung eines rekursiv definierten Nichtterminals vernünftig definiert ist. Das generelle Prinzip, rekursiven Definitionen Bedeutungen zuzuschreiben wird im Kapitel 6.4 bei der Fixpunktsemantik

besprochen.

Beispiel: Eine BNF–Grammatik für Summen–Ausdrücke

G1 = (N,∑,P,s)

N = {<exp>,<int>,<digit>}

∑ = {0,1,2,3,4,5,6,7,8,9,+}

P = {<exp> ::= <int> | <exp> + <exp>
<int> ::= 0 | <digit>{<digit> | 0}*
<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
}

s = <exp>

Bemerkungen:

- Die Definition von <int> erlaubt keine führenden Nullen.
- Normalerweise werden nur die Produktionen der Grammatik aufgelistet. Die Mengen der Nichtterminale und Terminale sind durch die gewählte Notation ersichtlich. Das zuerst definierte Nichtterminal ist das Startsymbol.

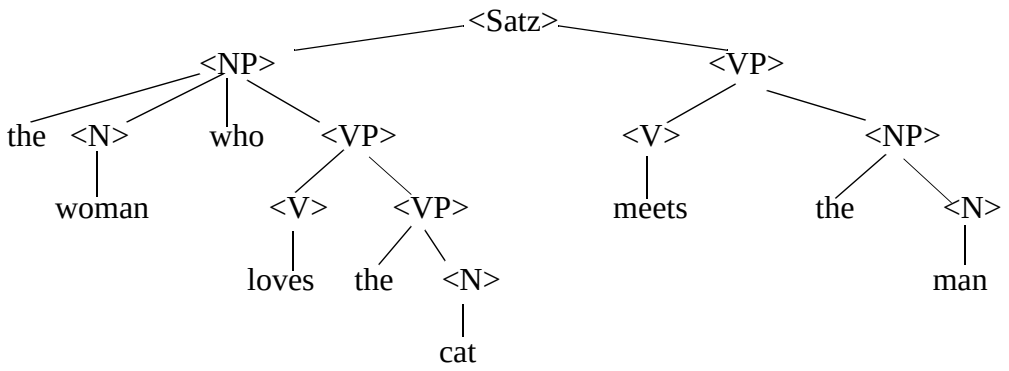
Die Zugehörigkeit eines Wortes zu einer formalen Sprache wird durch seine **Herleitung** nachgewiesen. Eine Herleitung wird mit einem Ableitungsbaum dargestellt.

Definition: (Ableitungsbaum, parse tree)

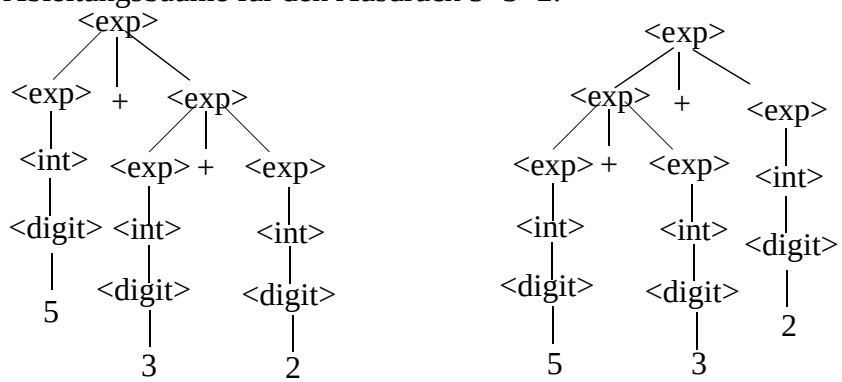
Ein **Ableitungsbaum** ist ein Baum mit den folgenden Eigenschaften:

(1) Alle Blätter sind Terminale.
(2) Alle Knoten sind Nichtterminale.
(3) Die Wurzel ist das Startsymbol.
(4) Die Nachfolger eines Knotens (Nichtterminals) müssen durch eine der Produktionen der Grammatik entstehen. Dazu wird gewählt: Eine der Alternativen der Vereinigung, eine Instanz der Iteration bzw. Option.

Beispiel: (1) Ableitungsbaum für den Satz „the women who loves the cat meets the man“



(2) Ableitungsbäume für den Ausdruck 5+3+2:



Im allgemeinen ist es der Fall, daß verschiedene Ableitungsbäume verschiedene Interpretationen desselben Wortes zulassen. Dies führt zum Begriff der Mehr-deutigkeit.

Definition: (mehrdeutig)

(1) Eine **Grammatik**, die es erlaubt, mindestens ein Wort mit mehreren verschiedenen Ableitungsbäumen herzuleiten, heißt **mehrdeutig**.

(2) Eine **Sprache**, die sich **nur** durch mehrdeutige Grammatiken definieren läßt, heißt auch **mehrdeutig**.

Man möchte, wenn irgend möglich, Mehrdeutigkeiten vermeiden, um die Portabilität der Sprache zu gewährleisten. Es ist nicht nur möglich, dasselbe Wort mit mehreren Ableitungsbäumen herzuleiten. Einunddieselbe Sprache kann auch mit mehreren Grammatiken definiert werden.

Beispiel: Eindeutige BNF–Grammatiken für Summen–Ausdrücke

Zwei weitere Grammatiken definieren <exp> folgendermaßen:

G2: <exp> ::= <int>
| <int> + <exp>

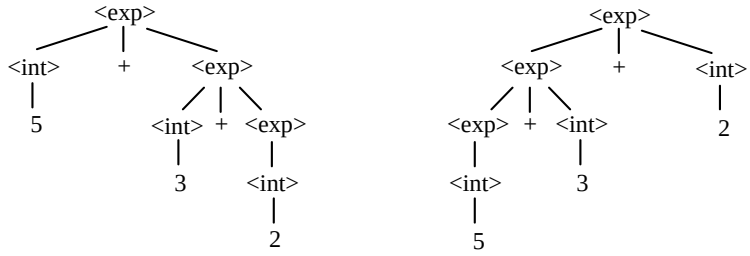
G3: <exp> ::= <int>

| <exp> + <int>

mit <int> wie gehabt.

Dann gilt: L(G1) = L(G2) = L(G3).

Es stellt sich heraus, daß G2 und G3 eindeutige Grammatiken sind. G2 läßt nur den folgenden linken Baum zu, G3 nur den folgenden rechten Baum:



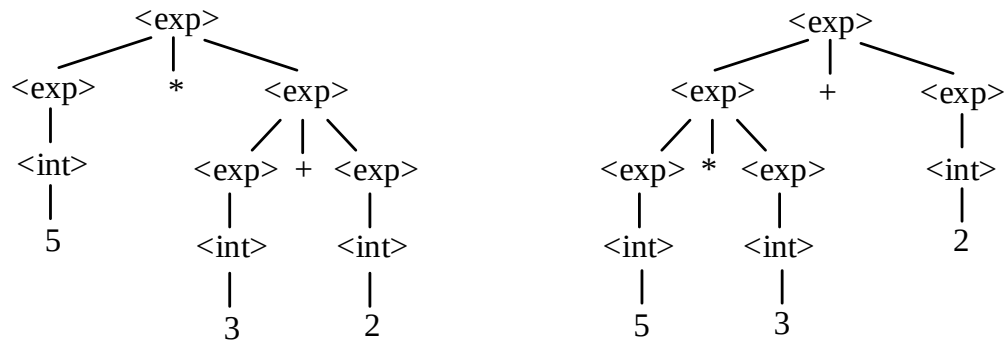
Im Beispiel der Summen–Ausdrücke führt die Mehrdeutigkeit von G1 nicht zu Schwierigkeiten, da die Addition assoziativ ist. (G2 schreibt Rechts–, G3 Links–Assoziativität vor. Dies ändert sich aber, wenn man weitere Operationen auf den ganzen Zahlen zuläßt.

Beispiel: BNF–Grammatiken für arithmetische Ausdrücke

G4: <exp> ::= <int>

| <exp> [+ | – | * | div] <exp>

Beachte: div ist ein Symbol, es besteht nicht aus mehreren Symbolen: div $\in \Sigma$. G4 läßt für den Ausdruck 5 * 3 + 2 die beiden folgenden Herleitungen zu:



Die entsprechenden Auswertungen ergeben:

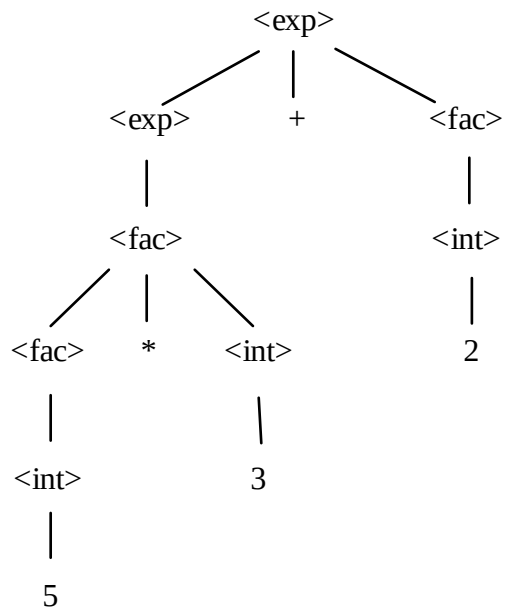
3 + 2 = 5 5 * 3 = 15

5 * 5 = 25 15 + 2 = 17

Das Problem ist, daß G4 es erlaubt, zu jeder Zeit jeden beliebigen Operator zu generieren. Mathematische Konvention schreibt aber die Auswertung von $*$ und div vor der von + und – vor. D.h. + und – müssen höher im Baum liegen, d.h. vor $*$ und div generiert werden. Dies läßt sich durch die Einführung eines neuen Nichtterminals erreichen.

G5: <exp> ::= <fac>
| <exp> [+ | –] <fac>
<fac> ::= <int>
| <fac> [∗ | div] <int>

G5 erlaubt nur die gewünschte Ableitung:



Bemerkung:

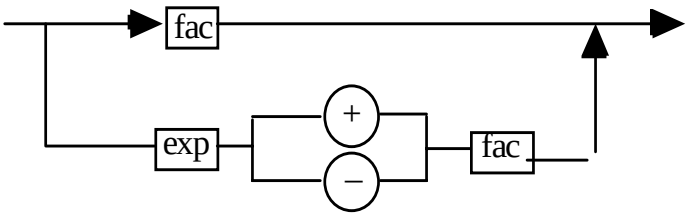
Mehrdeutigkeit und Präzedenz sind Konzepte, die das Verhalten des Programms betreffen. Trotzdem werden sie durch syntaktische Entscheidungen bestimmt! Oft werden Eigenschaften von Programmiersprachen, die prinzipiell syntaktisch erfaßt werden, aus Effizienzgründen nicht syntaktisch festgelegt. So führt die Elimination von Mehrdeutigkeiten im allgemeinen zu einer erhöhten Anzahl von Nichtterminalen, d.h. Produktionen, was die Spracherkennung (parsing) aufwendiger macht. Auch werden viele Programmiersprachen, die nicht kontextfrei sind, mit kontextfreien Grammatiken beschreiben. Weitere Einschränkungen werden dann durch sog. Kontextbedingungen spezifiziert [Broy, Kap.3.1.3].

6.1.2 Syntaxdiagramme

Syntaxdiagramme sind eine verwandte Form der Beschreibung formaler Sprachen.
Für jedes Nichtterminal gibt es ein separates Diagramm.

Beispiel: Syntaxdiagramme für arithmetische Ausdrücke von ganzen Zahlen.

exp :



= Nichtterminalsymbol

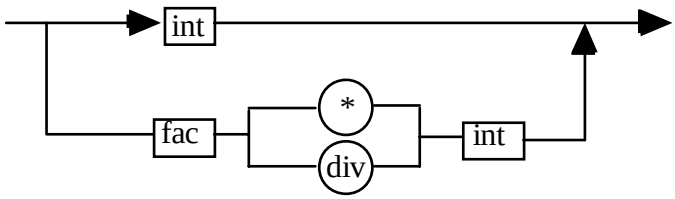
= Terminalsymbol

= Vereinigung

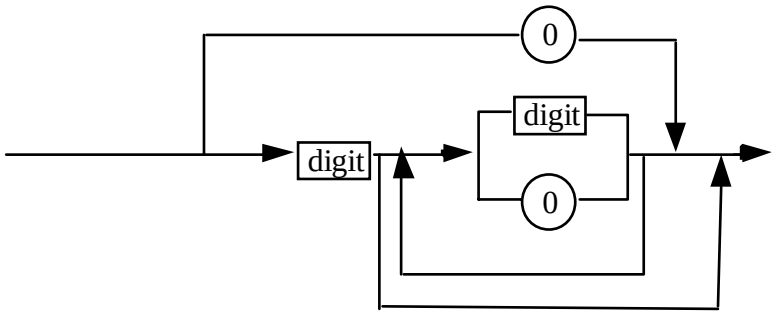
Hintereinandersetzen = Konkatenation

Schleife = Kleene–Star

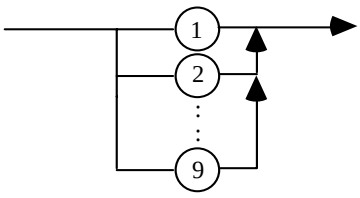
fac :



int :



digit :



6.2 Typüberprüfung

In diesem Abschnitt betrachten wir das Problem der Überprüfung des Typs eines SML-Ausdrucks anhand einer kleinen Untermenge von SML.
Wir stellen z.B. folgende Fragen:
Warum ist

`not true: bool`? Da `not:bool → bool, true: bool`

`f x: int` ? Gilt z. B., falls `f: int → int, x: int`

`(f(f x)): int` ? falls `f: int → int, x: int`

`if (not b) then (f x) else(f(f x)): int`
Gilt z.B., falls `b: bool` und `f: int → int`

`fn f => fn x => if (not b) then (fx) else f(f x)`
`: (int → int) → (int → int)`
Gilt, falls `b: bool`

6.2.1 Die funktionale Teilsprache

Die gewählte Teilsprache hat folgende kontextfreie Syntax:

<Typ> : : = <TypVar> |
<TypKonst> |
<Typ> → <Typ> |
(<Typ>)

<Ausdruck> : := <Id> | <Konstante>
<Ausdruck> <Ausdruck> |
if <Ausdruck> then <Ausdruck> else <Ausdruck> |
fn <Id> => <Ausdruck> |
(<Ausdruck>)

Als Typkonstanten wählen wir die Zahlentypen <TypKonst> :: = int | real | bool.
Als Konstante wählen wir die Boolschen Werte und einstellige Operationen sowie Zahlen und deren einstellige Operationen (für mehrstellige Operationen siehe 6.2.3).

```
true, false: bool,
not: bool → bool
0, 1, ...: int
0.0, ...: real
...
```

Beispiele:

a) Typen:
```
bool, int, 'a,
(int → int) → int,
bool → ('a → int)
```

b) Ausdrücke: `0, 0.0, 1, x, f, b`

```
(not b),
(f x),
f(f x),
if (not b) then (f x) else (f(f x))
fn x => 7
fn f => fn x => if(not b) then (f x) else (f(f x))
fn x => x
```

6.2.2 Typisierungsregeln

In diesem Abschnitt behandeln wir die Frage, ob ein Ausdruck E einen geg. Typ T besitzt. Im Allgemeinen hängt dies von den im Kontext angegebenen Typen der Identifikatoren ab.

Definition:

Ein **Kontext** ist eine endliche Menge der Form

$\{x_1: T_1, ..., x_n: T_n\}$, $n \geq 0$

wobei $x_1, ..., x_n$ Identifikatoren und $T_1, ..., T_n$ Typen sind.

Wir schreiben

$\Gamma \vdash E : T$

für die Aussage, daß der Ausdruck E den Typ T besitzt, wenn der Kontext bekannt ist.

Beispiel:

$\varnothing \vdash$ +: int*int $\rightarrow$ int
$\varnothing \vdash$ (not true): bool
$\varnothing \vdash$ fn x => 7: int $\rightarrow$ int
$\varnothing \vdash$ fn x => 7: (bool $\rightarrow$ bool) $\rightarrow$ int
x: int, f: int $\rightarrow$ int $\vdash$ (f x): int

Um Typisierungsaussagen abzuleiten, geben wir Axiome der Form

$\Gamma \vdash E : T$

und Regeln der Form

$$\frac{P_1 \ ... \ P_k}{Q}$$

wobei $P_1, ..., P_k$, Q Typisierungsaussagen sind.

Ein **Beweis** hat die Form eines Baums, so daß

(1) die Blätter Axiome sind
(2) die Wurzel die abgeleitete Typisierungsaussage
(3) für jeden inneren Knoten K gibt es eine Regel, so daß K Instanz der Konklusion und die Prämissen Instanzen der Kinder sind.

Im Gegensatz zu den üblichen Baumnotationen werden die Wurzel unten und die Blätter oben geschrieben.

Typisierungsregeln:

(1) Axiome (Regeln ohne Prämissen)

[konst] $\varnothing \vdash$ c: T, falls c eine Konstante vom Typ T

Beispiele:

$\varnothing \vdash 0: int$,
$\varnothing \vdash 0.0: real$,
$\varnothing \vdash true: bool$,
$\varnothing \vdash not: bool \to bool$

[var] $\{x: T\} \vdash x:T$ für bel. <Typ> T; <Id> x

Beispiele:

$\{x: int\} \vdash x: int$,
$\{x: a'\} \vdash x:'a$,
$\{f: int \to int\} \vdash f: int \to int$

(2) Regeln

[addhyp] $\dfrac{\Gamma \vdash E : T}{\Gamma \cup \{x: T1\} \vdash E: T}$ x nicht in Γ

$$\underbrace{\{f: int \to int, x: int\}}_{\Gamma \cup \{x: int\}} \vdash f: int \to int \quad \text{(aus } f: int \to int \vdash f: int \to int\text{)}$$

$$\frac{\varnothing \vdash 0: int}{\{x: int\} \vdash 0: int} \quad [addhyp]$$

Allgemein gilt:

$$\frac{\varnothing \vdash 0: int}{\Gamma \vdash 0: int}$$ [mehrere addhyp] für bel. Γ

[app1] $$\frac{\Gamma \vdash F: T \to U, \quad \Gamma \vdash E: T}{\Gamma \vdash F\ E: U}$$

Beispiel: Sei $\Gamma \vdash = \{f: int \to int, x: int, b = bool\}$

$$\frac{\Gamma \vdash f: int \to int \qquad \Gamma \vdash x: int}{\Gamma \vdash f\ x: int}$$

$$\frac{\Gamma \vdash f\ x: int \qquad \Gamma \vdash f: int \to int}{\Gamma \vdash f(f\ x): int}$$

$$\frac{\Gamma \vdash b: bool, \quad \Gamma \vdash not: bool\ bool}{\Gamma \vdash (not\ b): bool}$$

[if] $$\frac{\Gamma \vdash B: bool, \quad \Gamma \vdash E_1: T, \quad \Gamma \vdash E_2: T}{\Gamma \vdash \text{if } B \text{ then } E_1 \text{ else } E_2: T}$$

Beispiel: Sei $\Gamma = \{b: bool, f: int \to int, x: int\}$

$$\frac{\Gamma \vdash (not\ b): bool \quad \Gamma \vdash f\ x: int \quad \Gamma \vdash f(f\ x): int}{\Gamma \vdash \text{if } (not\ b) \text{ then } f\ x \text{ else } f(f\ x)}$$

[abs] $$\frac{\Gamma \cup \{x: T\} \vdash E: U}{\Gamma \vdash fn\ x => E : T \to U}$$ falls x nicht aus Γ

Beispiel:

$$\frac{\{x: int\} \vdash 7: int}{\varnothing \vdash fn\ x => 7 : int \to int}$$

$\Gamma_1 = \{f: int \to int, b: bool\}$

[abs] $$\frac{\Gamma_1 \cup \{x: int\} \vdash \text{if } (not\ b) \text{ then } (f\ x) \text{ else } (f\ x)}{\Gamma_1 \vdash fn\ x => \text{if } (not\ b) \text{ then } (f\ x) \text{ else } f(f\ x)}$$

$$\text{II} \qquad\qquad : (int \to int)$$

$\Gamma_0 \cup \{f: int \to int\}$, wobei $\Gamma_0 = \{b: bool\}$.

ochmalige Anwendung von [abs] ergibt:

$$\Gamma_0 \vdash fn\ f \Rightarrow fn\ x \Rightarrow \text{if } (not\ b) \text{ then } (fx) \text{ else } f(fx): (int \to int) \to (int \to int)$$

Ableitung polymorpher Typen
Bei jeder Ableitung kommt es darauf an, die Axiome richtig zu wählen.
Beispiel:
(1)

$$\dfrac{[\text{var}]\quad \{x: 'a\} \vdash x: 'a}{[\text{abs}]\quad \varnothing \vdash (\text{fn } x \Rightarrow x): a \Rightarrow 'a}$$

(2) ABER

$$\dfrac{[\text{var}]\quad \{x: 'a\} \vdash x: 'a \qquad [\text{abs}]\quad \varnothing \vdash \text{fn } x \Rightarrow x: \text{int} \to \text{int},\ \varnothing \vdash 3: \text{int}}{[\text{app}]\quad \varnothing \vdash (\text{fn } x \Rightarrow x)\ 3 \ : \text{int}}$$

6.2.3 Mehrstellige Funktionen und Überladen

Wir erweitern die Typen um Produkttypen

<Typ>: := <Typ>*<Typ>

Für 2-stellige Funktionen mit Infix gibt es folgende Regel:

$$[\text{app2}]\quad \frac{\Gamma \vdash F: T_1 * T_2 \to U,\quad \Gamma \vdash E_1: T_1,\quad \Gamma \vdash E_2: T_2}{\Gamma \vdash E_1\ F\ E_2: U}$$

Beispiel: Sei $\Gamma = \{f: \text{int} \to \text{int},\ x: \text{int}\}$

$$\dfrac{\dfrac{\Gamma \vdash +: \text{int} * \text{int} \to \text{int} \quad \Gamma \vdash (f\ x): \text{int} \quad \Gamma \vdash 0: \text{int}}{\Gamma \vdash (f\ x) + 0 : \text{int}} \qquad \Gamma \vdash +: \text{int} * \text{int} \to \text{int} \quad \Gamma \vdash 0: \text{int}}{\Gamma \vdash (0 + 0): \text{int}}$$

Z.B. das Symbol + ist überladen, d.h. + erlaubt (mindestens) 2 Typen mit den Axiomen

$\varnothing \vdash +: \text{int} * \text{int} \to \text{int}$

$\varnothing \vdash +: \text{real} * \text{real} \to \text{real}$

Damit können wir z.B. ableiten:

$$\dfrac{\varnothing \vdash +: \text{int} * \text{int} \to \text{int},\ \varnothing \vdash 0: \text{int},\ \varnothing \vdash 1: \text{int}}{\varnothing \vdash 0 + 1: \text{int}}$$

$$\dfrac{\varnothing \vdash +: \text{real} * \text{real} \to \text{real},\quad \varnothing \vdash 0.0: \text{real},\ \varnothing \vdash 1.0: \text{real}}{\varnothing \vdash 0.0 + 1.0: \text{real}}$$

6.2.4. Algorithmus zur Bestimmung des allgemeinsten Typs eines Ausdrucks[1]

Mit dem Typinferenzsystem der vorhergehenden Abschnitte läßt sich die Frage beantworten, ob ein SML-Ausdruck einen bestimmten Typ besitzt. Im Folgenden geben wir einen Algorithmus an, der es erlaubt zu jedem (korrekten) SML-Ausdruck den allgemeinsten Typ zu berechnen.
Die Grundlage dazu bilden die Begriffe der Substitution und Unifikation.

a) Substitution und Unifikation

Eine (Typ-) Substitution σ ist eine Abbildung von Typvariablen in Typausdrücke, wobei nur an endlich vielen Stellen $\sigma(\alpha) \neq \alpha$ gilt.

Beispiel:

σ = [int/α, bool/β, $\gamma \rightarrow \gamma/\delta$] bedeutet:
$\sigma(x) = x$ für $x \neq \alpha,\beta,\delta$;
$\sigma(\alpha)$ = int, $\sigma(\beta)$ = bool, $\sigma(\delta) = \gamma \rightarrow \gamma$.

Definition:

Zwei Typ-Ausdrücke s_1, s_2 heißen *unifizierbar*, wenn es eine Substitution σ gibt mit

$\sigma^*(s_1) = \sigma^*(s_2)$,

wobei σ^* die Erweiterung von σ auf Typausdrücke bezeichnet:

$\sigma^*(\alpha) = \sigma(\alpha)$ für α aus dem Definitionsbereich von σ,
$\sigma^*(\alpha) = \alpha$ sonst,
$\sigma^*(s \rightarrow s') = \sigma^*(s) \rightarrow \sigma^*(s')$,

σ heißt in diesem Fall Unifikator von s_1 und s_2.

Ein Unifikator *mgu* von s_1 und s_2 heißt *allgemeinster Unifikator*, falls für jeden Unifikator σ von s_1 und s_2 gilt:

Es gibt eine Substitution φ mit $\varphi \circ mgu = \sigma$.
d.h. $\varphi(mgu(s)) = \sigma(s)$ für alle Ausdrücke s.

Beispiel:

Unifiziere $\alpha \rightarrow \alpha$ mit $(\beta \rightarrow \text{int}) \rightarrow \gamma$:
Die Unifikation von α mit $\beta \rightarrow$ int ergibt $\alpha = \beta \rightarrow$ int.
Daraus erhält man $\gamma = \beta \rightarrow$ int, d.h.
σ = [$\beta \rightarrow$ int/α, $\beta \rightarrow$ int/γ] ist (allgemeinster) Unifikator von $\alpha \rightarrow \alpha$ und $(\beta \rightarrow \text{int}) \rightarrow \gamma$.
Weitere Unifikatoren sind:
σ_1 = [bool $\rightarrow$ int/α, bool $\rightarrow$ int/γ] oder
σ_2 = [(int $\rightarrow$ int) $\rightarrow$ int/α, (int $\rightarrow$ int) $\rightarrow$ int/γ].
Es gilt $\sigma_1 = \varphi_1 \circ \sigma$ mit φ_1 = [bool/β] und

1. Dieser Abschnitt ist weiterführend und gehört nicht zum Kernstoff der Vorlesung.

$\sigma_2 = \varphi_2 \circ \sigma$ mit $\varphi_2 = [\text{int} \to \text{int}/\beta]$.

Bemerkung: Wir schreiben im Folgenden häufig für Substitutionen Gleichungen $\alpha = s$ anstelle von s/α.

Beispiel (für eine nicht erfolgreiche Unifikation)

Unifikation von $\alpha \to \alpha$ mit $(\beta \to \text{int}) \to \beta$, ergibt zunächst $\alpha = (\beta \to \text{int})$. Daraus folgt für den Resultattyp: $\beta = \beta \to \text{int} = (\beta \to \text{int}) \to \text{int} = \ldots$

Da β sowohl auf der linken wie auf der rechten Seite der Ungleichung auftritt ("Occur check"), gibt es keine (endliche) Lösung von $\beta = \beta \to \text{int}$:

$\alpha \to \alpha$ und $(\beta \to \text{int}) \to \beta$ sind nicht unifizierbar.

Satz (Robinson 1965)

Für je zwei Typausdrücke s_1 und s_2 gilt:

Entweder sind s_1 und s_2 nicht unifizierbar oder es gibt einen allgemeinsten Unifikator von s_1 und s_2.

Beweis durch Angabe des Algorithmus (**Unifikationsalgorithmus von Robinson**)

Seien s_1 und s_2 Typausdrücke.

1.) $s_1 \equiv \alpha$:

1) α tritt in s_2 auf: Dann sind s_1 und s_2 nicht unifizierbar,

2) α tritt nicht in s_2 auf: Dann $\text{mgu}(s_1, s_2) = [s_2/\alpha]$.

2.) $s_2 \equiv \alpha$: Symmetrisch zu 1.).

3.) $s_1 \equiv c$ oder $s_2 \equiv c$ (Konstante). Dann müssen s_1 und s_2 beide gleich c sein,
ansonsten sind s_1 und s_2 nicht unifizierbar.

4.) $s_1 \equiv T_1 \to U_1$:

Falls $s_2 \equiv T_2 \to U_2$ berechne

a) $\varphi = \text{mgu}(T_1, T_2)$,

b) Instanziiere U_1, U_2 mit φ und berechne $\sigma = \text{mgu}(\varphi(U_1), \varphi(U_2))$.

Dann ist $\sigma \circ \varphi$ allgemeinster Unifikator von s_1 und s_2.

Ansonsten sind s_1 und s_2 nicht unifizierbar.

Beispiel:

(1) $s_1 \equiv \alpha \to (\alpha \to \text{bool})$, $s_2 \equiv (\text{int} \to \beta) \to \gamma$.

Unifikation:

[1] Unifikation von α und $\text{int} \to \beta$ ergibt $\varphi = [\text{int} \to \beta/\alpha]$,

[2] Unifikation von $\varphi(\alpha \to \text{bool}) = (\text{int} \to \beta) \to \text{bool}$ und $\varphi(\gamma) = \gamma$ ergibt $\sigma = [(\text{int} \to \beta) \to \text{bool}/\gamma]$.

Also $\text{mgu}(s_1, s_2) = \sigma \circ \varphi = [\text{int} \to \beta/\alpha, (\text{int} \to \beta) \to \text{bool}/\gamma]$.

(2) $(\alpha \to \text{int})$ und $(\alpha \to (\beta \to \beta)$ sind nicht unifizierbar, da $\text{int} \neq \to$.

(3) $s_1 \equiv \alpha \to (\alpha \to \alpha)$, $s_2 \equiv \beta \to \beta$.

[1] Unifikation von α und β ergibt $\varphi = [\beta/\alpha]$,

[2] Unifikation von $\varphi(\alpha \to \alpha) = \beta \to \beta$ und $\varphi(\beta) = \beta$ ergibt Berechnung von $\beta \to \beta$: Dies ist nicht unifizierbar, da durch Ersetzen von β durch $\beta \to \beta$ ein Zyklus entsteht (Occur check).

Also sind s_1 und s_2 nicht unifizierbar.

mb) Typbestimmungsalgorithmus

Auf Eingabe eines Ausdrucks bzw. einer Deklaration e prüft der Algorithmus, ob e typisierbar ist und gibt den allgemeinsten Typ von e aus.

Im folgenden bezeichnen α_1 , ..., α_n, β Typvariablen und s, s_1, s_2, t_1, t_2, ... , t_n Typausdrücke.

Bemerkung: Generell gilt: Bei der Bestimmung von Typen müssen, wenn immer möglich, neue Typvariablen eingeführt werden. Bereits erzielte Zwischenergebnisse müssen bei der weiteren Berechnung verwendet werden.

<< Der Algorithmus fehlt noch>>

6.3. Semantik

Es gibt verschiedene Arten von Semantiken, die sich im "Auflösungsvermögen", d.h. in der Detailliertheit ihrer Beschreibungen unterscheiden. Sehr detaillierte Semantiken geben dem Implementierer viel Information, sind aber für den Programmierer zu umständlich. Zum Programmieren gibt es gröbere Semantiken.

–	Axiomatische Semantik	**Programmierer**
:		:
:		:
–	Funktionale (denotationelle) Semantik	:
–	Operationelle Semantik	**Implementierer**

(1) Axiomatische Semantik:

Eine axiomatische Semantik beruht auf **Axiomen**, die die Bedeutung der einzelnen Sprachkonstrukte definieren. D.h. jede elementare Programmeinheit (z.B. Wertzuweisungen oder das leere Programm) und jeder Programmkombinator (z.B. Komposition, Alternation, Wiederholung) werden axiomatisch definiert. Durch Anwendung der Axiome kann der Programmierer vom Verhalten der einzelnen Sprachkonstrukte auf das Verhalten des Programms schließen. Das Hoare–Kalkül von Vor– und Nachbedingungen ist eine axiomatische Semantik. Wir werden eine Hoare–Semantik für imperative Programme kennenlernen.

(2) Funktionale Semantik:

Eine funktionale Semantik ist eine **Abbildung** der syntaktischen Repräsentation einer Programmiersprache in ein semantisches Modell. Das semantische Modell hat im wesentlichen die Aufgabe, das Ein/Ausgabe-Verhalten jedes Programms in der betreffenden Programmiersprache festzulegen. Die Modellierung erfolgt durch eine Abbildung, die im allgemeinen aus vielen Abbildungen zusammengesetzt sein kann. Im semantischen Modell kann man auch Rechnerkonzepte wie Speicher, Scope, Parameterübergabe, etc. auf abstrakte Weise beschreiben. Damit kann eine funktionale Semantik schon konkrete Hinweise auf eine Implementation liefern. Eine Stärke funktionaler Semantik ist, daß für sie eine Fülle von relativ einfach formulier- und beschreibbaren Theoremen zur Verfügung steht.

(3) Operationelle Semantik:

Eine operationelle Semantik beschreibt nicht nur das "Was", sondern auch das "Wie" des Programmverhaltens. Die Grundelemente eines operationellen Modells sind Rechnerkonzepte wie **Zustand** und **Zustandsänderung** (**Transition**). Das Programmverhalten wird als eine Menge von möglichen Transitionsfolgen dargestellt. Für viele Sprachen wird zunächst eine operationelle Semantik definiert. [Wik, Kap.10] gibt Auszüge einer operationellen Semantik von ML. Die komplette formale Syntax und Semantik von ML findet sich in dem (100-seitigen) Buch [Milner].

Es wird empfohlen, für eine Programmiersprache zwei Semantiken von möglichst unterschiedlicher Auflösung anzugeben, und ihre Verträglichkeit zu beweisen. Dazu müssen zwei Dinge gezeigt werden:

– **Konsistenz**: Jedes legale Verhalten in der feinen Semantik ist auch ein legales Verhalten in der groben Semantik, soweit dort beschreibbar.

– **Vollständigkeit**: Jedes legale Verhalten in der groben Semantik ist auch ein legales Verhalten in der feinen Semantik.

Wir wenden uns im folgenden nur der funktionalen Semantik zu. Eine funktionale Semantik besteht aus einer Interpretationsabbildung I, einer Syntaxdomäne Prog und einem semantischen Modell Mod:

$$I: Prog \rightarrow Mod$$

Wir spezifizieren eine funktionale Semantik in drei Etappen:

(1) **Die Syntaxdomäne**. Wir definieren die Syntaxdomäne durch eine Grammatik. Wir benennen auch die Sprachen, die von den Nichtterminalen erzeugt werden.

(2) **Die semantischen Rechenstrukturen**. Wir geben die Rechenstrukturen an, aus

denen sich das semantische Modell aufbaut.

(3) **Die Interpretationsabbildung**. Wir definieren I, indem wir eine Abbildung per Nichtterminal angeben. I ist dann die Abbildung des Startsymbols.

6.3 Semantik von Rekursion: Fixpunkte

In den Beispielsemantiken, die wir durchgesprochen haben, traten keine rekursiven Sprachkonstrukte auf, wie es sie z.B. in ML gibt. Wenn wir etwa das Konstrukt

```
let val rec I = E1 in E2 end;
```

semantisch definieren wollen, so müssen wir, wenn wir die Bedeutung von I bestimmen, in E1 auf gerade diese Bedeutung zurückgreifen. Dies führt zu einer unendlichen Auswertung. Um sie zu vermeiden, muß die Rekursion irgendwie "gebrochen" werden. (Das Schlüsselwort `rec` ist eigens eingeführt worden, um ML vor dieser Auswertung zu bewahren.)

Betrachten wir die Problematik allgemeiner – losgelöst von ML.

6.3.1 Einführung

Eine Definition kann als eine Gleichung angesehen werden. Die Bedeutung der Definition ist durch die Lösung(en) der Gleichungen gegeben.

Beispiele:

(1) x = 3
Die Lösung dieser Gleichung in der Unbekannten x ist 3, denn Substitution von 3 für x macht die Gleichung wahr.

(2) x = x + 1
Diese Gleichung hat keine Lösung und ist daher als Definition unbrauchbar.

(3) x = x
Diese Gleichung hat sehr viele Lösungen. Welche soll als Definition für x gelten?

Beispiele (2) und (3) sind rekursive Gleichungen. Rekursive Gleichungen brauchen keine Lösung zu haben, können aber auch sehr viele Lösungen haben. Als Definition kommen nur Gleichungen in Frage, die mindestens eine Lösung haben. Wenn mehrere Lösungen existieren, muß eine als definierende Lösung ausgezeichnet werden.

Kommen wir nun zu Funktionsdefinitionen. Was ist die Bedeutung der folgenden

definierenden Gleichung?

$q: \mathbf{N} \to \mathbf{N}^{\perp}$ wobei $\mathbf{N}^{\perp} =_{def} \mathbf{N} \cup \{\perp\}$

(*) $$q(n) = \begin{cases} 1, & \text{falls } n = 0 \\ q(n+1) & \text{sonst} \end{cases}$$

d.h. (*) ist eine Menge von Gleichungen mit der Unbekannten q; die Gleichungen müssen für dasselbe q für alle $n \in \mathbf{N}$ gelten.

Was wissen wir über die Lösungen von Gleichung (*)?

– Sie müssen Argument 0 auf 1 abbilden.
– Für alle Argumente n, die nicht null sind, muß das Bild dasselbe sein, wie das des Nachfolgers von n.

Eine große Anzahl von Funktionen erfüllt diese Kriterien, genau gesagt: alle Funktionen in der Menge

$\{(fn\ n \Rightarrow \textbf{if } n=0 \textbf{ then } 1 \textbf{ else } k) \mid k \in N^{\perp}\}$

d.h. die Menge

$$\{f_k \mid f_k(n) = \begin{cases} 1, & \text{falls } n = 0 \\ k & \text{sonst} \end{cases}, k \in N^{\perp}\}$$

Wir können nicht alle Funktionen als Bedeutung der Gleichung (*) zulassen, sondern müssen uns für eine entscheiden.

Die Theorie der Lösung solcher Gleichungen heißt **Fixpunkttheorie**. Warum? Gesucht ist eine Lösung für die Unbekannte q in Gleichung (*). Machen wir q zu einem zusätzlichen Parameter der rechten Seite unserer Gleichung, so erhalten wir das **Fixpunkt-Funktional** Q ("Funktional", da es Funktionen auf Funktionen abbildet):

$Q: (\mathbf{N} \to \mathbf{N}^{\perp}) \to (\mathbf{N} \to \mathbf{N}^{\perp})$

$$Q(q)(n) = \begin{cases} 1, & \text{falls } n = 0 \\ q(n+1), & \text{falls } n > 0 \end{cases}$$

d.h.

$Q(q) = h$ mit $h(0) = 1$, $h(n) = q(n+1)$ für alle $n > 0$

Die Lösungen von Gleichung

(*) $q(n) = Q(q)(n)$ für alle $n \in \mathbf{N}$,

bzw. $q = Q(q)$

sind genau die Funktionen h, die unter Q auf sich selbst abgebildet werden: Q[h] = h. ([Broy] schreibt das Funktionsargument eines Funktionals in eckigen Klammern.)
Um eine Lösung einer rekursiven Gleichung zu finden, muß:

– mindestens ein Fixpunkt existieren,
– wenn mehrere Fixpunkte existieren, einer als die Lösung ausgewählt werden,
– die Lösung konstruierbar, d.h. berechenbar sein.

Es stellt sich heraus, daß Bedingungen für das Funktional angegeben werden können, unter denen ein Fixpunkt existiert. Wenn mehrere existieren, wird traditionsgemäß der **kleinste Fixpunkt (least fixed point)** bzgl. einer bestimmten Ordnung auf Funktionen gewählt. In dieser Ordnung, der sog. Approximationsordnung, ist eine Funktion f kleiner als eine Funktion g, wenn f mit g übereinstimmt, aber unter Umständen an weniger Punkten definiert ist.

Definition: (Funktionsiteration)
Eine Funktion $f : M \to M$ kann wie folgt **iteriert** werden:

$f^0 =_{def} c$, wobei c eine Konstante ist;

$f^{i+1} =_{def} f(f^i)$.

Man kann nun das Fixpunktfunktional iterieren und erhält dadurch Annäherungen an die Lösung der Gleichung, für die das Funktional steht. Für die folgenden Beispiele brauchen wir noch ein Element, das Undefiniertheit im Raum der Funktionen repräsentiert.

Definition: (Überall nichtdefinierte Funktion)
Die **überall nichtdefinierte Funktion** (über einer beliebigen Menge M) wird folgendermaßen bezeichnet:

$\Omega: M \to M^{\perp}$ mit

$\Omega(n) = \perp$ für alle $n \in M$.

In unserem obigen Beispiel ergibt sich also folgende Situation:

(1) Definition des Funktionals Q:
Sei die Gleichung
$q = Q(q)$
definiert durch

$$q(n) = \underbrace{\begin{cases} 1, & \text{falls } n = 0 \\ q(n+1), & \text{sonst} \end{cases}}_{Q(q)(n)}$$

d.h. das Funktional Q enthält zwei Parameter: q, n.
Q hat den Typ

$Q: (\mathbf{N} \to \mathbf{N}^{\perp}) \to (\mathbf{N} \to \mathbf{N}^{\perp})$

In SML wird Q implementiert durch

```
val Q = fn q => fn n => if n = 0 then 1 else q(n +1)
```

(2) Lösung durch Approximation

Wir definieren die Funktionaliteration $(Q^i)_{i \in \mathbf{N}}$:

Start mit „total undefinierter" Fkt Ω:

$Q^0 = \Omega$ (d.h. $Q^0: \mathbf{N} \to \mathbf{N}^{\perp}$)

$Q^{i+1} = Q(Q^i)$ (d.h. $Q^i : \mathbf{N} \to \mathbf{N}^{\perp}$)

$Q^{\infty} = \lim_{i \to \infty} Q^i$

$$\text{wobei } Q^1(n) = Q(Q^0)n = \begin{cases} 1 & \text{falls } n = 0 \\ Q^0(n+1) & \text{sonst} \end{cases} = \begin{cases} 1 & \text{falls } n = 0 \\ \perp & \text{sonst} \end{cases}$$

$$Q^{i+1}(n) = Q(Q^i)n = \begin{cases} 1 & \text{if } n = 0 \\ \underbrace{Q^i(n+1)}_{\perp \text{ (da } Q^i(m) = \perp \text{ für } m > 0)} & \text{sonst} \end{cases}$$

$$\text{Also } Q^{\infty}(n) = \begin{cases} 1 & \text{if } n = 0 \\ \perp & \text{sonst} \end{cases}$$

Die Frage ist, unter welchen Voraussetzungen der Grenzwert existiert. In diesem Fall gilt, daß Q^{∞} existiert und zu $M \to M^{\perp}$ gehört; der Grenzwert wird schon vom zweiten Element (und allen weiteren Elementen) der Folge $(Q^i)_{i \in \mathbf{N}}$ angenommen.

Der Limes liefert eine Lösung der Gleichung

$q = Q(q)$

Denn es gilt

$$Q(Q^{\infty})(n) = \begin{cases} 1 & \text{falls } n = 0 \\ Q^{\infty}(n+1) & \text{sonst} \end{cases} = \begin{cases} 1 & \text{falls } n = 0 \\ \perp & \text{sonst} \end{cases} = Q^{\infty}(n)$$

d.h. Q^{∞} ist Lösung der Gleichung. Daß Q^{∞} auch die kleinste Lösung ist, folgt aus dem Satz von Kleene (siehe später).

6.3.2 Theoretische Grundlagen der Fixpunkttheorie

Kommen wir nach diesen einführenden Beispielen zur Fixpunktheorie selbst. Wir müssen zunächst eine partielle Ordnung definieren, bzgl. der der auserwählte Fixpunkt am kleinsten

ist.

Definition: (Partielle Ordnung)

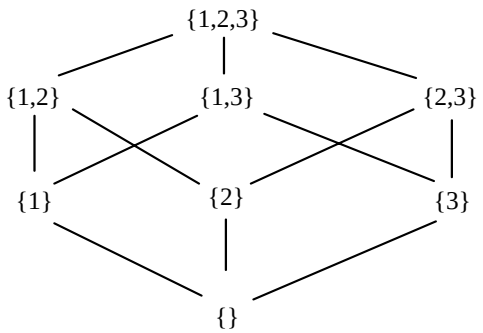
Eine Relation $\leq \subseteq M \times M$ heißt **partielle Ordnung** gdw.

1. $(\forall\ x : x \in M : x \leq x)$ Reflexivität
2. $(\forall\ x,y : x,y \in M : x \leq y \wedge y \leq x \Rightarrow x=y)$ Antisymmetrie
3. $(\forall\ x,y,z : x,y,z \in M : x \leq y \wedge y \leq z \Rightarrow x \leq z)$ Transitivität

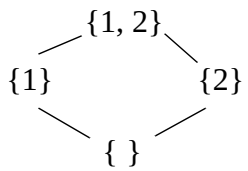
Beispiele:

(1) Mengeninklusion auf der Potenzmenge von {1,2,3}.

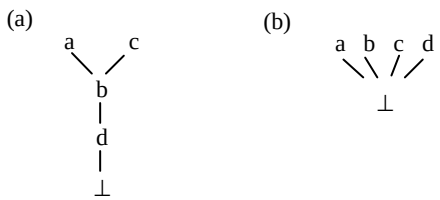
Eine partiell geordnete Menge kann durch einen azyklischen Graphen repräsentiert werden.



(2) Mengeninklusion auf {1, 2}



(3) Partielle Ordnungen auf der Menge $\{a,b,c,d\}^{\perp}$.



Sei in den folgenden Definitionen ≤ eine partielle Ordnung auf M.

Definition: (Bottom)

$\perp$ (genannt **Bottom**) ist das Element mit der folgenden Eigenschaft:

$$(\forall\, x \in M : \perp \leq x)$$

Falls es existiert, ist $\perp$ das kleinste Element in M bzgl. $\leq$

Definition: (Flache Ordnung)

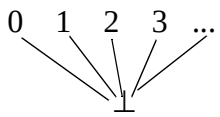
Es existiere ein kleinstes Element $\perp$. Wenn die partielle Ordnung $\leq$ der folgenden Definition genügt:

$$x1 \leq x2 \quad \Leftrightarrow \quad x1 = \perp \vee x1 = x2$$

so heißt $\leq$ die **diskrete Ordnung** (oder auch die **flache** Ordnung)

Die diskrete Ordnung induziert punktweise eine Appoximationsordnung auf Funktionen.

Beispiel:



$N^{\perp}$

$N^{\perp}$ ist eine flache Ordnung mit:

$$x1 \leq x2 \quad \Leftrightarrow \quad x1 = \perp \text{ oder } x1 = x2.$$

Bemerkung:

Aus typographischen Gründen schreiben wir im folgenden „$\leq$“ für alle Ordnungen, d. h. auch für die flache Ordnung und die Approximationsordnung. (In der Vorlesung wurden dafür eckige Klammern verwendet.)

Definition: (Approximationsordnung)

Sei $\leq$ die diskrete Ordnung auf M und seien f,g: $N \rightarrow M$ Funktionen. Dann ist die Relation

$$f \leq g \quad \Leftrightarrow \quad (\forall\, x \in N .\ f(x) \leq g(x))$$

eine partielle Ordnung, die sog. **Approximationsordnung**. f ist eine **Approximation** von g.

Bemerkung:

Falls M flach, gilt $f \leq g$ gdw $\forall\, x \in N .\ f(x) = \perp$ oder $f(x) = g(x)$

Definition: (gerichtete Menge; obere Schranke; Kette; abzählbare Kette)

Eine nichtleere Teilmenge $X \subseteq M$ heißt **gerichtet**, gdw.

$$(\forall\ x,y \in X . (\exists\, z \in X . x \leq z \wedge y \leq z))$$

z heißt **obere Schranke** von x und y.

Eine nichtleere Teilmenge $X \subseteq M$ heißt **Kette** gdw.

$$(\forall\, x,y \in X . x \leq y \vee y \leq x)$$

Wenn X abzählbar ist, kann man es darstellen als $X = \{x_i \mid i \in N\}$, sodaß gilt:

$$(\forall\ i \in \mathbf{N} \,.\, x_i \leq x_{i+1})$$

Man schreibt eine abzählbare Kette auch $(x_i)_{i \in \mathbf{N}}$.

Definition: (Supremum)

Sei $X \subseteq M$. Das Supremum von x; oder kürzer sup X, ist das Element von M (falls es existiert), für das gilt:

(1) $(\forall\ x \in X \,.\, x \leq \sup X)$

(2) $(\forall\ y \in M \,.\, (\forall\ x : x \in X : x \leq y) \Rightarrow (\sup X) \leq y)$

sup X heißt das **Supremum** (die kleinste obere Schranke) von X.

Definition: (Vollständige Ordnung, complete partial order, cpo)

Eine Menge M heißt **vollständig geordnet** (complete partial order, cpo) gdw.

$$(\forall\ X \subseteq M \,.\, X \text{ gerichtet} \Rightarrow (\sup X) \in M)$$

Beispiele: Seien M, N vollständig geordnet

- Die Funktionenmenge $M \to N$ ist bezüglich der Approximationsordnung vollständig geordnet.
- Jede Menge ist bzgl. der diskreten Ordnung vollständig geordnet.
- **N** ist bzgl.der üblichen ≤ Ordnung **nicht** vollständig geordnet.
- $\mathbf{N} \cup \{\infty\}$ ist bzgl. der üblichen ≤ Ordnung vollständig geordnet.

Unsere Lösungsapproximationen sollen eine Kette im Raum der Funktionen bilden. Dazu brauchen wir den Begriff der Funktionsmonotonie.

Definition: (Funktionsmonotonie)

Eine Funktion $f: M \to N$ ist **monoton** gdw.

$$x \leq y \quad \Rightarrow \quad f(x) \leq f(y)$$

Wir verlangen die Monotonie von unserem Funktional F, d.h.

$$f \leq g \quad \Rightarrow \quad F(f) \leq F(g).$$

Satz: Wenn F monoton ist, dann ist die Folge $(F^i)_{i \in \mathbf{N}}$ eine Kette, wobei $F^0 =_{def} \bot$, $F^{i+1} =_{def} F(F^i)$ für $i \geq 0$.

Beweis: (Induktion über i)

Basis: $F^0 = \Omega$; Ω ist das kleinste Element im Raum $M \to M^{\bot}$.

Induktionsannahme: Sei $F^{i-1} \leq F^i$.

Induktionsschritt: Wegen der Monotonie von F:

$$F^i = F(F^{i-1}) \leq F(F^i) = F^{i+1}$$ qed.

Wir möchten als Lösung für unsere rekursive Gleichung das Supremum unserer Approximationsfolge wählen:

$\sup\{F^i \mid i \in \mathbf{N}\}$

Um dies zu erreichen, müssen wir fordern, daß das Funktional F Suprema in Suprema überführt.

Definition: (Funktionsstetigkeit)

Eine Funktion f: M $\rightarrow$ N ist (ketten)stetig gdw.

$f(\sup\{x_i \mid i \in \mathbf{N}\}) = (\sup\{f\, x_i \mid i \in \mathbf{N}\})$

Beispiel: (1) Eine nicht monotone Funktion.

halt: $\mathbf{N}^{\perp} \rightarrow \mathbf{B}$

halt($\perp$) = false

halt(x) = true

Es gilt $\perp \leq x$ aber halt($\perp$) = false, aber true = halt(x).

false und true sind in der diskreten Ordnung nicht vergleichbar.

(2) Eine nicht-stetige Funktion

all: $(\mathbf{N} \rightarrow \mathbf{N}^{\perp}) \rightarrow \mathbf{B}$

$$\text{all}(f) = \begin{cases} \text{true} & \text{falls } f(a) = 0 \text{ für alle } a \in \mathbf{N} \\ \text{false} & \text{falls } f(a) > 0 \text{ für ein } a \in \mathbf{N} \\ \perp & \text{sonst} \end{cases}$$

Betrachte die Kette

$$f_i(a) = \begin{cases} 0 & \text{falls } a \leq i \\ \perp & \text{sonst} \end{cases}$$

$(\sup f_i)(a) = 0$ für alle i

Es gilt $\text{all}(f_i) = \perp$ für alle i, d.h. $\sup\{\text{all}(f_i) \mid i \in \mathbf{N}\} = \perp$

aber $\text{all}(\sup(f_i)) = 0$.

6.3.3. Der Fixpunktsatz von Kleene

Satz: (Fixpunktsatz von Kleene)

Ist F eine stetige Abbildung über einer vollständigen geordneten Menge M mit kleinstem Element $\perp$, so ist die Gleichung

$x = F(x)$

lösbar und es existiert ein **eindeutiger** kleinster Fixpunkt fix F von F, der wie folgt konstruiert wird:

fix F = sup{F^i | i ∈ **N**}

Beweis:

(1) fix F ist ein Fixpunkt von F:

F(fix F)

= F(sup{F^i | i ≥ 0})

= {Stetigkeit von F}

sup{F(F^i) | i ≥ 0}

= {Umformen}

sup{F^i | i ≥ 1}

= {$F^0 = \perp$ und $\perp \leq F(\perp)$}

sup{F^i | i ≥ 0}

= fix F

(2) fix F ist kleiner als jeder andere Fixpunkt:

Sei x∈ M ein Fixpunkt von F, d.h. x = F(x). Da $\perp \leq x$ und F monoton, folgt

(1) $F^0 = \perp \leq x$

(2) Ind.hyp $F^i \leq x$

F monoton $\Rightarrow F^{i+1} = F(F^i) \leq F(x) = x$

Also fix F = sup F^i x

D.h. fix F = {F^i | i ∈ **N**} ≤ x.

qed.

Bemerkungen:

- Die **Existenz** eines eindeutigen Fixpunktes läßt sich schon garantieren, wenn man nur die Monotonie von F voraussetzt (Satz von Knaster–Tarski [Broy]). Für das konstruktive Berechnungsverfahren durch Iteration des Fixpunktfunktionals F braucht man allerdings dessen Stetigkeit.
- Die Tatsache, daß wir den kleinsten Fixpunkt als Lösung für rekursive Gleichungen gewählt haben, garantiert uns nicht nur seine rechnerische Approximierbarkeit, sondern auch größtmögliche Freiheiten für den Implementierer, da wir die Lösung gewählt haben, die an den wenigsten Stellen festgelegt ist.

Beispiel:

(1) Betrachten Sie folgende Funktion

even: $\mathbf{N} \rightarrow \mathbf{B}^{\perp}$

mit rekursiver Definition:

$$\text{even}(n) = \begin{cases} \text{true} & \text{falls } n = 0 \\ \text{false} & \text{falls } n = 1 \\ \text{even}(n-2) & \text{sonst} \end{cases}$$

d.h. das Funktional

E: $(\mathbf{N} \rightarrow \mathbf{B}^{\perp}) \rightarrow (\mathbf{N} \rightarrow \mathbf{B}^{\perp})$

ist definiert durch

$$E(f)(n) = \begin{cases} \text{true} & \text{falls } n = 0 \\ \text{false} & \text{falls } n = 1 \\ f(n-2) & \text{sonst} \end{cases}$$

Die Funktionaliteration ergibt:

$$E^1(n) = \begin{cases} \text{true} & \text{falls } n = 0 \\ \text{false} & \text{falls } n = 1 \\ \underbrace{E^0(n-2)}_{\perp} & \text{sonst} \end{cases}$$

$$E^2(n) = \begin{cases} \text{true} & \text{falls } n = 0 \\ \text{false} & \text{falls } n = 1 \\ E^0(n-2) & \text{sonst} \end{cases} = \begin{cases} \text{true} & \text{falls } n = 0 \\ \text{false} & \text{falls } n = 1 \\ \text{true} & \text{falls } n = 2 \\ \text{false} & \text{falls } n = 3 \\ \perp & \text{sonst} \end{cases}$$

$$E^i(n) = \begin{cases} \text{true} & \text{falls } n < 2i,\ n = \text{gerade} \\ \text{false} & \text{falls } n < 2i,\ n = \text{ungerade} \\ \perp & \text{sonst} \end{cases}, \quad i \geq 1$$

$$\underbrace{E^{\infty}(n)}_{(\text{fix } E)(n)} = \begin{cases} \text{true} & \text{falls } n = \text{gerade} \\ \text{false} & \text{falls } n = \text{ungerade} \end{cases}$$

(2) Sei die Gleichung gegeben

$$f(n) = \begin{cases} 1 & \text{falls } n \leq 0 \\ n * f(n-1) & \text{sonst} \end{cases}$$

d.h. $f = F(f)$

wobei das Funktional F definiert ist als

$$F(f)(n) =_{def} \begin{cases} 1 & \text{falls } n \leq 0 \\ n * f(n-1) & \text{sonst} \end{cases} \qquad \text{für alle } f \in \mathbf{N} \to \mathbf{N}^{\perp},\ n \in \mathbf{N}$$

Funktionaliteration:

$$F_0 =_{def} \Omega$$

$$F^i =_{def} F(F^i)$$

Also

$$F_0(n) = \perp \qquad \text{für alle } n \in \mathbf{N}$$

$$F_1(n) = \begin{cases} 1 & \text{falls } n \leq 1 \\ n * \underbrace{F_0(n)}_{\perp} & \text{sonst} \end{cases} = \begin{cases} 1 & \text{falls } n \leq 1 \\ \perp & \text{sonst} \end{cases}$$

$$F_2(n) = \begin{cases} 1 & \text{falls } n \leq 1 \\ n * F_1(n-1) & \text{sonst} \end{cases} = \begin{cases} 1 & \text{falls } n \leq 1 \\ 2 * \underbrace{F_1(1)}_{1} & \text{falls } n = 2 \\ n * \underbrace{F_1(n-1)}_{\perp} & \text{falls } n > 2 \end{cases} = \begin{cases} 1 & \text{falls } n \leq 1 \\ 2 & \text{falls } n = 2 \\ \perp & \text{sonst} \end{cases}$$

$$F_3(n) = \begin{cases} 1 & \text{falls } n \leq 1 \\ n * F_2(n-1) & \text{sonst} \end{cases} = \begin{cases} 1 & \text{falls } n \leq 1 \\ 2 * F_2(1) & \text{falls } n = 2 \\ 3 * F_2(2) & \text{falls } n = 3 \\ n * \underbrace{F_2(n-1)}_{\perp} & \text{falls } n > 3 \end{cases} = \begin{cases} 1 & \text{falls } n \leq 1 \\ 2 & \text{falls } n = 2 \\ 6 & \text{falls } n = 3 \\ \perp & \text{sonst} \end{cases}$$

d.h. für $i \geq 1$

$$F_i(n) = \begin{cases} n! & \text{falls } n \leq i \\ \perp & \text{sonst} \end{cases}$$

Daraus folgt für den Limes:

$$F^{\infty}(n) = (\sup\{F_i \mid i \in \mathbf{N}\})(n)$$
$$= \sup\{\underbrace{F_i(n)}_{n!} \mid i \in \mathbf{N}\} = n!$$

F^{∞} löst die Gleichung (*):

$$F(F^{\infty})(n) = \begin{cases} 1 & \text{falls } n \leq 1 \\ n * F^{\infty}(n-1) & \text{sonst} \end{cases} = \begin{cases} 1 & \text{falls } n \leq 1 \\ \underbrace{n * (n-1)!}_{n!} & \text{sonst} \end{cases} = n! = F^{\infty}(n)$$

Die Elemente der Folge $(Fac^i)_{i \in \mathbf{N}}$ sind mit wachsendem i an mehr und mehr Stellen definiert. Der Grenzwert wird erst im Unendlichen erreicht, d.h. kein Element der Folge beschreibt das gesamte Verhalten der Lösung Fac^{∞}. Nach dem Satz von Kleene ist Fac^{∞} der kleinste Fixpunkt des Funktionals F.

6.4 Denotationelle Semantik

Durch die denotationelle Semantik wird jedem (korrekt gebildeten) Typausdruck s eine Menge von Werten M zugewiesen. Außerdem erhält jeder typkorrekte Ausdruck e des Typs s ein Element $a \in M$ als seine Bedeutung zugewiesen.
Die Bedeutung eines Ausdrucks bezeichnen wir im folgenden durch Klammern **[.]**, d.h. die Bedeutung eines Typausdrucks s ist die Menge **[**s**]**, die Bedeutung eines typkorrekten Ausdrucks e ist **[**e**]**. Ist e vom Typ s, so gilt **[**e**]** $\in$ **[**s**]** oder **[**e**]** $= \perp_{[s]}$.

Beispiel:

Der Typ bool der Wahrheitswerte erhält die Bedeutung **[**bool**]** = {T, F}, die Bedeutung von true ist **[**true**]** = T, die von not(true) ist **[**not(true)**]** = F.

6.4.1 Bedeutung von Typausdrücken (ohne Typvariable)

Jeder Typausdruck s erhält als Bedeutung eine Menge, die mit **[**s**]** bezeichnet wird.

a) Typkonstanten:
[bool**]** $=_{def}$ {T, F},
[int**]** $=_{def} = Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

b) Außerdem definieren wir allgemein die Erweiterung einer Menge M durch $\perp_M$:
$M^{\perp} =_{def} M \cup \{\perp_M\}$. Ist $M \neq N$, so gilt $\perp_M \neq \perp_N$.
Falls keine Mißverständnisse auftreten können, lassen wir häufig den Index von $\perp_M$ weg und schreiben $\perp$ anstelle von $\perp_M$.

c) Für zwei Typausdrücke s_1, s_2 ohne Typvariablen
Sei

$[\![s_1 \to s_2]\!] =_{def} [\![s_1]\!] \to_{stetig} [\![s_2]\!]^{\perp}$

wobei $M \to_{stetig} N$ die Menge der stetigen Funktionen von M nach N bezeichnet.

Sind M, N cpo's, so ist $M \to_{stetig} N^{\perp}$ ein cpo mit kleinstem Element

$\Omega: M \to N^{\perp}, \quad \Omega(m) = \perp$ für alle $m \in M$.

d) Wir bezeichnen im folgenden die Vereinigung aller Bedeutungen von Typausdrücken (ohne Typvariablen) mit D:

$D =_{def} \{x \in [\![s]\!]^{\perp} \mid s$ Typausdruck ohne Typvariablen$\}$.

Die Interpretation jeder Standardoperation f wird mit f^D bezeichnet.

6.4.2 Bedeutung der Standardoperationen und Ausdrücke

a) Jede Standardkonstante c:s erhält einen Wert $c^D \in [\![s]\!]$, der also verschieden von $\perp_{[\![s]\!]}$ ist.

Beispiel:

$true^D = T$, $0^D = 0$

b) Für jede Standardoperation $f: s_1 \to s_2$ fordern wir, daß sie eine totale Funktion $f^D: [\![s_1]\!] \to [\![s_2]\!]^{\perp}$ bezeichnet.

Beispiel:

$not^D: B \to B^{\perp}$ mit
$not^D(T) = F$, $not^D(F) = T$.

Beispiel:

$+^D: (Z \times Z) \to Z^{\perp}$.
$+^D(a, b) = a + b$ für $a, b \in Z$.

c) if – then – else:

Die Bedeutung des if – then – else Konstrukts wird durch die Funktion

$if^D: B^{\perp} \times M^{\perp} \times M^{\perp} \to M^{\perp}$, mit
$if^D(\perp_B, a, b) = \perp_M$,
$if^D(T, a, b) = a$,
$if^D(F, a, b) = b$
für alle $a, b \in M$,

bestimmt.

d) Ausdrücke:

Für die Semantik eines Ausdrucks f(e) mit einer Standardoperation $f: s_1 \to s_2$ berechnet man zunächst den Wert a von e. Falls $a \in [\![s_1]\!]$, so berechnet man anschließend den Wert $f^D(a)$ der Anwendung von f^D auf a; falls $a = \perp \notin [\![s_1]\!]$, so ist $[\![f(e)]\!] = \perp$.

Um diesen Zusammenhang allgemein zu formulieren, müssen wir die Werte der in e vorkommenden Identifikatoren (Namen) kennen. Wir definieren zunächst die Menge ENV der Umgebungen env:

$ENV = \{env: ID \to D \mid env(x) \in [\![s]\!]^{\perp}$ für $x:s\}$.

Dann ist die Interpretation $[\![e]\!]_{env}$ eines Ausdruckes e in der Umgebung env $\in$ ENV ein Element aus D. Wir definieren:

d.1) $[\![f]\!]_{env} =_{def} f^D$ für jede Standardkonstante oder Standardoperation f,

d.2) $[\![x]\!]_{env} =_{def} env(x)$ für jeden Identifikator x,

d.3) $[\![\text{if b then } e_1 \text{ else } e_2]\!]_{env} =_{def} if^D([\![b]\!]_{env}, [\![e_1]\!]_{env}, [\![e_2]\!]_{env})$.

d.4) Die Funktionsapplikation wird folgendermaßen definiert:

Seien $e_1:s_1 \to s_2$, $e_2:s_1$.

$[\![(e_1\ e_2)]\!]_{env} =_{def} [\![e_1]\!]_{env}([\![e_2]\!]_{env})$, falls $[\![e_1]\!]_{env} \in [\![s_1 \to s_2]\!]$ und $[\![e_2]\!]_{env} \in [\![s_1]\!]$,

$[\![(e_1\ e_2)]\!]_{env} =_{def} \perp$ sonst.

Beispiel: Es gilt:

$[\![not(true)]\!]_{env} = [\![not]\!]_{env}([\![true]\!]_{env}) = not^D(T) = F.$

Für env mit env(x) = F gilt

$[\![not(x)]\!]_{env} = [\![not]\!]_{env}([\![x]\!]_{env}) = not^D(env(x)) = not^D(F) = T.$

d.5) Die Funktionsabstraktion wird "punktweise" definiert.

Sei $s_1 \to s_2$ der Typ von fn x $\Rightarrow$ e.

$[\![\text{fn x} \Rightarrow e]\!]_{env} =_{def}$ diejenige Funktion h mit h: $[\![s_1]\!] \to [\![s_2]\!]^{\perp}$ und $h(a) = [\![e]\!]_{env1}$, wobei $env1(x) =_{def} a$ und $env1(z) =_{def} env(z)$ für $z \neq x$.

Beispiel:

(1) $[\![\text{fn (x:int)} \Rightarrow x]\!]_{env} = h_1:Z \to Z^{\perp}$ mit $h_1(a) = a$.

(2) $[\![\text{fn (x:int)} \Rightarrow 0]\!]_{env} = h_2:Z \to Z^{\perp}$ mit $h_2(a) = 0$.

Es gilt: $[\![\text{fn (x:int)} \Rightarrow 0)\ 3]\!]_{env} = 0$,

$[\![\text{fn (x:int)} \Rightarrow 0)\ (7 \text{ div } 0)]\!]_{env} = \perp$, weil $[\![7 \text{ div } 0]\!]_{env} = \perp \notin Z$.

Bemerkung: Die Bedeutung eines Ausdruckes e hängt nur von den in e auftretenden Identifikatoren ab. Enthält e keine freien Identifikatoren, so gilt $[\![e]\!]_{env1} = [\![e]\!]_{env2}$ für alle Umgebungen env_1 und env_2. Die Bedeutung von e hängt dann nicht von der Umgebung ab. Wir definieren dann:

$[\![e]\!] =_{def} [\![e]\!]_{env}$.

Wir sind eigentlich nur an der Bedeutung von Ausdrücken ohne freie Identifikatoren interessiert, die Umgebungen benötigen wir als Hilfskonstruktionen. Ausdrücke mit freien Identifikatoren braucht man im Rumpf von Funktionen.

d.6) Die Semantik einer rekursiven Deklaration

Sei die Deklaration

fun f x $\Rightarrow$ e

mit x: s1, e: s2 im Kontext Γ gegeben. Sei env eine zugehörige Umgebung. Die Semantik wird definiert über die Semantik des Funktionals.

Φ = fn f $\Rightarrow$ (fn x $\Rightarrow$ e): $(s_1 \to s_2) \to (s_1 \to s_2)$

$[\![\Phi]\!]_{env} \in ([\![s_1]\!] \to [\![s_2]\!]^{\perp}) \to_{stetig} ([\![s_1]\!] \to [\![s_2]\!]^{\perp})$

$[\![\Phi]\!]_{env}(g)(y) =_{def} [\![e]\!]_{env2}$

wobei env2(f) $=_{def}$ g, env2(x) $=_{def}$ y,

env2(z) $=_{def}$ env(z) für z $\neq$ f und z $\neq$ y

ist ein stetiges Funktional, das einen Fixpunkt

$(fix[\![\Phi]\!]_{env})$: $[\![s1]\!] \to [\![s2]\!]^{\perp} = [\![s1 \to s2]\!]$

besitzt.

Also definieren wir

$[\![fun\ f\ x \Rightarrow e]\!]_{env} =_{def} fix[\![\Phi]\!]_{env}$

Beispiel: Test auf gerade Zahl:

[fun even n =

if n = 0 then true

else if n = 1 then false

else even (n - 2)

$]\!]_{env} = \Phi^{\infty}: \mathbf{N} \to \mathbf{B}^{\perp}$

mit

$$\Phi^{\infty}(x) = \begin{cases} T & \text{falls x gerade, } x \geq 0 \\ F & \text{falls x ungerade, } x \geq 0 \\ \perp & \text{falls } x < 0 \end{cases}$$

Denn

Φ = fn f $\Rightarrow$ fn n $\Rightarrow$ if n = 0 then true

else if n = 1 then false

else f(n - 2)

Dies ergibt das Funktional

$[\![\Phi]\!]_{env}$: $[\![int \to bool]\!] \to_{stetig} [\![int \to bool]\!] = (\mathbf{N} \to \mathbf{B}^{\perp}) \to_{stetig} (\mathbf{N} \to \mathbf{B}^{\perp})$

definiert durch

$$[\![\Phi]\!]_{env}(g)(y) = \begin{cases} T & \text{falls } y = 0 \\ F & \text{falls } y = 1 \\ g(y - 2) & \text{sonst} \end{cases}$$

Dann sei Φ^{∞} das Supremum der Funktionaliteration

$\Phi^0 = \Omega$, $\Phi^{i+1} = [\![\Phi]\!]_{env}(\Phi^i)$.

Wir zeigen als Probe, daß Φ^{∞} Fixpunkt von $[\![\Phi]\!]_{env}$ ist:

$[\Phi]_{env}(\Phi^{\infty})(y) = [\text{Def. } [\Phi]_{env}]$

$$\begin{cases} T & \text{falls } y = 0 \\ F & \text{falls } y = 1 \\ \Phi^{\infty}(y - 2) & \text{sonst} \end{cases} = [\text{Def. } \Phi^{\infty}, \text{ alg. Umformung}]$$

$$\begin{cases} T & \text{falls } y = 0 \\ F & \text{falls } y = 1 \\ T & \text{falls } y \text{ gerade, } y \geq 2 \\ F & \text{falls } y \text{ ungerade, } y \geq 2 \\ \perp & \text{falls } x < 0 \end{cases} = [\text{ alg. Umformung}]$$

$$\begin{cases} T & \text{falls } y \text{ gerade, } y \geq 0 \\ F & \text{falls } y \text{ ungerade, } y \geq 0 \\ \perp & \text{falls } x < 0 \end{cases} = \Phi^{\infty}(y)$$