

Formale Grundlagen der Informatik 3 –

5. Syntax und Semantik von \mathcal{L} -Programmen

Christoph Walther
TU Darmstadt

1 Syntax von \mathcal{L}

1.1 Datentypen in \mathcal{L}

Definition 1 (Allgemeine Form einer Datentypdefinition)

Datentypen werden in \mathcal{L} definiert durch Ausdrücke der Form:

$$\begin{aligned}
 \text{structure struc}[@T_1, \dots, @T_n] <= & \\
 \text{cons}_1(\text{sel}_{1,1}: \text{struc}_{1,1}, \dots, \text{sel}_{1,n_1}: \text{struc}_{1,n_1}), & \\
 \vdots & \\
 \text{cons}_k(\text{sel}_{k,1}: \text{struc}_{k,1}, \dots, \text{sel}_{k,n_k}: \text{struc}_{k,n_k}) &
 \end{aligned} \tag{1}$$

mit

- (1) $@T_1, \dots, @T_n$ ist eine (u.U. leere) Liste von paarweise verschiedenen *Typvariablen* (Notation bei leerer Liste: `structure struc <= ...` anstatt `structure struc[] <= ...`)
- (2) $k \geq 1$, d.h. es gibt mindestens *einen* (Daten-)Konstruktor,
- (3) $n_i \geq 0$ für alle $i \in \{1, \dots, k\}$, d.h. *Konstruktorkonstante* sind erlaubt,

- (4) für alle $i \in \{1, \dots, k\}$ und alle $h \in \{1, \dots, n_i\}$ ist $\text{struc}_{i,h}$ ein Datentyp mit
- (a) $\text{struc}_{i,h}$ ist *Instanz* eines *bereits definierten* Datentyps
oder es gilt $\text{struc}_{i,h} = \text{struc}[@T_1, \dots, @T_n]$
(= *rekursive Definition erlaubt*)
 - (b) $\text{struc}_{i,h} \neq \text{bool}$,
 - (c) in $\text{struc}_{i,h}$ werden höchstens die Typvariablen
aus $\{@T_1, \dots, @T_n\}$ verwendet,
 - (d) die Bezeichner struc , cons_i , $\text{sel}_{i,h}$ sind voneinander
verschieden und bislang nicht verwendet worden
- (5) für *ein* $i \in \{1, \dots, k\}$ und alle $h \in \{1, \dots, n_i\}$ ist
 $\text{struc}_{i,h}$ *verschieden von* $\text{struc}[@T_1, \dots, @T_n]$. ■

Dabei gilt:

Definition 2 (*Instanzen von Datentypen*)

Ein Datentyp struc' ist eine Instanz eines Datentyps struc gdw. struc' aus struc durch Ersetzen von Typvariablen in struc durch Datentypen und/oder Typvariable entsteht. ■

Damit:

- Jeder Datentyp ist Instanz von sich selbst.
- Monomorphe Datentypen besitzen keine Instanzen außer sich selbst.

Bedeutung von Definition 1:

Ein Ausdruck der Form (1) definiert

- ein Funktionssymbol eq_{struc} mit Signatur
 $\text{eq}_{\text{struc}} : \text{struc}[@T_1, \dots, @T_n] \times \text{struc}[@T_1, \dots, @T_n] \rightarrow \text{bool}$ und
- ein Funktionssymbol if_{struc} mit Signatur
 $\text{if}_{\text{struc}} : \text{bool} \times \text{struc}[@T_1, \dots, @T_n] \times \text{struc}[@T_1, \dots, @T_n] \rightarrow \text{struc}[@T_1, \dots, @T_n]$

sowie für jedes $i \in \{1, \dots, k\}$

- ein “neues” *Konstruktor*funktionssymbol cons_i mit Signatur
 $\text{cons}_i : \text{struc}_{i,1} \times \dots \times \text{struc}_{i,n_i} \rightarrow \text{struc}[@T_1, \dots, @T_n]$
- ein “neues” *Strukturprädikat*symbol $? \text{cons}_i$ mit Signatur
 $? \text{cons}_i : \text{struc}[@T_1, \dots, @T_n] \rightarrow \text{bool}$
- für jede Argumentposition $h \in \{1, \dots, n_i\}$ eines “neuen” Konstruktors
 cons_i ein “neues” *Selektor*funktionssymbol $\text{sel}_{i,h}$ mit Signatur
 $\text{sel}_{i,h} : \text{struc}[@T_1, \dots, @T_n] \rightarrow \text{struc}_{i,h}$

Zweck der Forderungen 1 – 4 von Definition 1:

- Übliche syntaktische Forderungen nach den Prinzipien
 - “nur Definiertes darf auch verwendet werden”,
 - “bereits Definiertes darf nicht erneut definiert werden”,
 - “bei Verwendung von bereits definierten Begriffen müssen deren syntaktische Forderungen respektiert werden”

Zweck von Forderung 5 von Definition 1:

- Verbietet rekursive Definitionen *ohne Rekursionsverankerung*
 - Beispiel: `structure void <= new(old:void)`

Bemerkung 1 *Bezeichner für Typvariable in VeriFun*

- beginnen immer mit @ (nur wg. optischer Unterscheidung),
- enthalten nur *alphanumerische* Zeichen.

Bemerkung 2 (*Vordefinierte Funktionen eq und if*)

- **Zweck:** Mit Definition eines neuen Datentyps stehen Funktionssymbole zur Repräsentation von *Gleichheit* und *bedingten Ausdrücken* zur Verfügung.
- **Besonderheiten:**
 - “ $\text{eq}_{\text{struc}}(x, y)$ ” wird im PrettyPrint von *VeriFun* als “ $x = y$ ” angezeigt
 - Bei Eingaben *muß* “ $x = y$ ” anstatt “ $\text{eq}_{\text{struc}}(x, y)$ ” geschrieben werden
 - Um Verwechslungen mit “ $=$ ” der Metasprache zu vermeiden wird “ eq_{struc} ” jedoch außerhalb von Beispielen und Screenshots verwendet

Bemerkung 3 (*Strukturprädikate*)

- $? \text{cons}_i(t)$ ist *Abkürzung* für $\text{eq}_{\text{struc}}(t, \text{cons}_i(\text{sel}_{i,1}(t), \dots, \text{sel}_{i,n_i}(t)))$
- *Beispiel:* “ $? :: (k)$ ” steht als *Abkürzung* für “ $k = \text{hd}(k) :: \text{tl}(k)$ ”
- *Also:* $? \text{cons}_i(t)$ gilt gdw. t mittels cons_i “darstellbar” ist

1.2 Prozeduren in \mathcal{L}

Definition 3 (Allgemeine Form einer Prozedurdefinition)

Prozeduren werden in \mathcal{L} definiert durch Ausdrücke der Form:

$$\mathbf{function\ proc}(x_1 : \mathbf{struc}_1, \dots, x_k : \mathbf{struc}_k) : \mathbf{struc} \leq \mathbf{body}_{\mathbf{proc}} \quad (2)$$

mit

- (1) $k \geq 1$, d.h. *Konstantensymbole* sind immer *Konstruktorsymbole*,
- (2) für alle $i \in \{1, \dots, k\}$ ist \mathbf{struc}_i Instanz eines bereits definierten Datentyps $\neq \mathbf{bool}$
- (3) der Bezeichner \mathbf{proc} ist bislang nicht verwendet worden
- (4) \mathbf{struc} ist Instanz eines bereits definierten Datentyps in dem höchstens Typvariable aus den \mathbf{struc}_i vorkommen
- (5) $\mathbf{body}_{\mathbf{proc}}$ (= *Prozedurrumpf*) ist ein Term vom Typ \mathbf{struc} über
 - den paarweise verschiedenen Variablensymbolen x_1, \dots, x_k (= *formale Parameter*) mit “Datentyp von x_i ist \mathbf{struc}_i ” und
 - den durch bereits definierte Datentypen und Prozeduren eingeführten Funktionssymbolen erweitert um \mathbf{proc} (= *rekursive Definition möglich*). ■

Bedeutung:

Ein Ausdruck der Form (2) definiert ein “neues” Prozedurfunktionssymbol `proc` mit Signatur $\text{proc} : \text{struc}_1 \times \dots \times \text{struc}_k \rightarrow \text{struc}$

Zweck der Forderungen 1 – 5 von Definition 3:

- Übliche syntaktische Forderungen nach den Prinzipien
 - “nur Definiertes darf auch verwendet werden”,
 - “bereits Definiertes darf nicht erneut definiert werden”,
 - “bei Verwendung von bereits definierten Begriffen müssen deren syntaktische Forderungen respektiert werden”

Zusätzliche Forderung:

- In Prozedurrümpfen dürfen *Bedingungen* in bedingten Ausdrücken *keine bedingten Ausdrücke* enthalten
- **Also:** `if{if{a, b, c}, d, e}` verboten. Schreibe statt dessen `if{a, if{b, d, e}, if{c, d, e}}` (Warum darf man das? => *Übung*)
- **Grund:** Nur wegen besserer Lesbarkeit

1.3 Erweiterung von \mathcal{L}

Vor Definition der Semantik von \mathcal{L} betrachten wir noch 3 nützliche Spracherweiterungen.

1.3.1 case-Ausdrücke

- Mit case-Ausdrücken werden *strukturelle* Fallunterscheidungen in Prozeduren und Lemmata modelliert
- Für einen Datentyp `struc` wie in Definition 1, einen Term t vom Typ `struc` und Terme r_1, \dots, r_n eines Typs `struc'` erhält man eine strukturelle Fallunterscheidung bzgl. `struc` durch

```

case t of
  cons1 : r1,
  cons2 : r2,
  ...
  consk : rk
end_case

```

(Prozedurale Schreibweise)

```

case { t ;
  cons1 : r1,
  cons2 : r2,
  ...
  consk : rk }

```

bzw.

(Funktionale Schreibweise)

- Ergebnisse, die für *mehrere* Konstruktoren gelten, können durch *other* *zusammengefaßt* werden:

```
case t of
  consi : ri,
  consj : rj,
  ...
  other : r
end_case
```

(Prozedurale Schreibweise)

bzw.

```
case {t ;
  consi : ri,
  consj : rj,
  ...
  other : r }
```

(Funktionale Schreibweise)

Bemerkung 4 (case-Ausdrücke)

Die Konstruktorfunktionssymbole in case-Ausdrücken dürfen in beliebiger Reihenfolge angegeben werden.

Beispiel 1 (*case-Ausdrücke*)

```
function #leaves(x:sexpr):nat <=
case x of
  nil   : 0,
  atom  : 1,
  cons  : #leaves(car(x)) + #leaves(cdr(x))
end_case
```

- berechnet die Anzahl der Blätter in einem Binärbaum.

```
function #nodes(x:sexpr):nat <=
case x of
  cons  : +( #nodes(car(x)) + #nodes(cdr(x)) ),
  other : 0
end_case
```

- berechnet die Anzahl der inneren Knoten in einem Binärbaum.

1.3.2 let-Ausdrücke

- Mit `let`-Ausdrücken können Terme an *lokale Variable* gebunden werden, um die *Mehrfachberechnung* eines Terms zu vermeiden
- `let`-Ausdrücke sind in *Prozeduren* und *Lemmata* erlaubt
- **Syntax:**

`let var := t in r end_let` *und/oder* `let{ var := t; r }`

mit

- *var* ist ein Variablensymbol eines Typs τ verschiedenen von den bislang verwendeten Funktionssymbolen, den formalen Parametern einer Prozedur bzw. den allquantifizierten Variablen eines Lemmas (in denen der *let*-Ausdruck vorkommt)
- *t* ist ein Term vom Typ τ (über der gegebenen Signatur), in dem *var* nicht vorkommen darf
- *r* ist ein Term (über der gegebenen Signatur), der auch Vorkommen von *var* enthalten darf (und sinnvollerweise auch enthält)

Beispiel 2 (*let-Ausdrücke*)

```
function depth(x:sexpr):nat <=
case x of
  cons : let car-depth := depth(car(x)) in
         let cdr-depth := depth(cdr(x)) in
         if car-depth > cdr-depth
         then +(car-depth)
         else +(cdr-depth)
         end_if
         end_let
         end_let,
  other : 0
end_case
```

- berechnet die Tiefe eines Binärbaums.

```
function minimum(k : list[nat]) : nat <=
if ?∅(k)
then 0
else if ?∅(tl(k))
then hd(k)
else let min-tl := minimum(tl(k)) in
      if hd(k) > min-tl
      then min-tl
      else hd(k)
      end_if
      end_let
      end_if
end_if
```

- berechnet das Minimum einer *nicht-leeren* Liste k von natürlichen Zahlen.

Elimination von **let**-Ausdrücken

- zu jedem **let**-Ausdruck

$\text{let } var := t \text{ in } r \text{ end_let}$ bzw. $\text{let}\{var := t; r\}$

existiert ein *let-freier* Term r' *gleicher Deutung*:

- r' entsteht aus r durch Ersetzung jedes Vorkommens von var in r durch t
 - funktioniert, da var nicht in t vorkommt
- In Beweiszielen *goal* einer *HPL*-Sequenz können **let**-Bindungen mittels der *HPL*-Regel *Purge* eliminiert werden
(\Rightarrow erhöht zuweilen die Lesbarkeit von Zieltermen im *Proof Viewer*)

1.3.3 Partiiell definierte Prozeduren

Für Prozeduren mit *monomorphem* Ergebnistyp können wir für *Argumente außerhalb des Definitionsbereichs* der zu berechnenden Funktion ein *willkürlich gewähltes Ergebnis* angeben¹, etwa

```
function minimum(k : list[nat]) : nat <=  
  if ?∅(k)  
    then 0  
    else ...  
  end_if
```

Aber:

- Willkürliches und unintuitives Ergebnis für $\text{minimum}(\emptyset)$!
- Was machen wir bei *polymorphen* Ergebnistypen ?

¹ Natürlich nur dann, wenn der *Definitionsbereich* der zu berechnenden Funktion *entscheidbar* ist (\Rightarrow Vorlesung “Berechenbarkeit”).

Beispiel:

```
function last(k : list[@T]) : @T <=  
if ?∅(k)  
  then ???  
  else if ?∅(tl(k))  
    then hd(k)  
    else last(tl(k))  
  end_if  
end_if
```

berechnet (von links gelesen) das letzte Element einer *nicht-leeren polymorphen* Liste k .

Frage: Wie definieren wir $\text{last}(\emptyset)$?

Problem: Wegen *Polymorphie* kann für $\text{last}(\emptyset)$ kein willkürlich gewähltes Ergebnis angegeben werden (denn dieses müßte vom Typ $@T$ sein) !

Lösung:

- Wir erlauben *einseitige* Fallunterscheidungen in Prozeduren
- *Notation:*
 - `if cond then term else * end_if` **statt** `if cond then term end_if`
 - `if cond then * else term end_if` **statt** `if \neg cond then term end_if`
 - für case-Ausdrücke entsprechend

Damit beispielsweise:

```
function last(k : list[@T]) : @T <=
if ?∅(k)
  then *
  else if ?∅(tl(k))
    then hd(k)
    else last(tl(k))
  end_if
end_if
```

Damit beispielsweise:

```
function log2(x : nat) : nat <=
if ?0(x)
  then ★
  else if ?0(¬(x))
    then 0
    else if even(x)
      then +(log2(half(x)))
      else ★
    end_if
  end_if
end_if
```

berechnet den binären Logarithmus von 2er-Potenzen, d.h. $\log_2(n) = m$

nur falls $n = 2^m$.

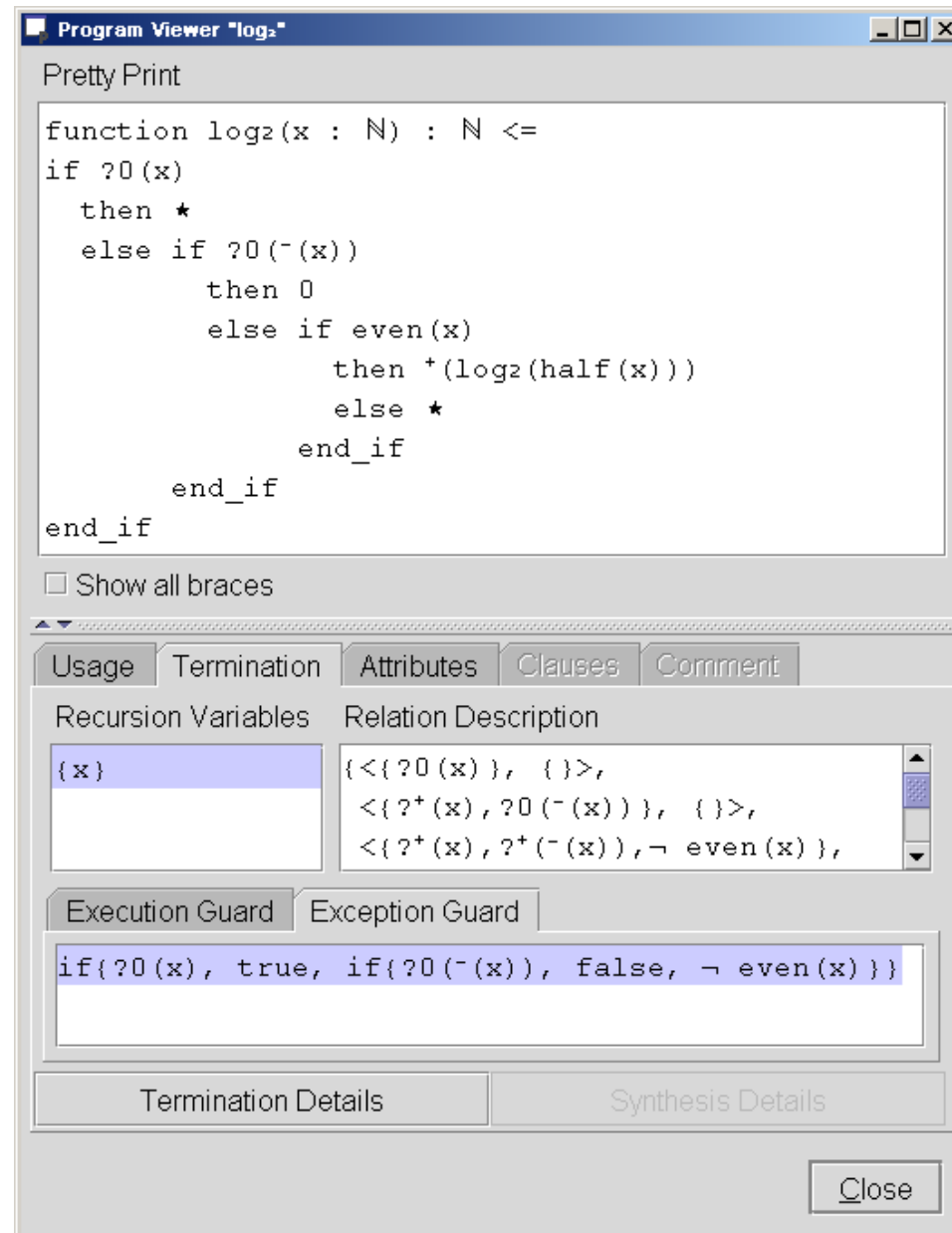
Die Exception Guard

- **Anschauung:** Bei Ausführung einer *Prozedur* mit aktuellen Parametern, die zu einem \star im Prozedurrumpf führen, wird eine *Exception* (= *Laufzeitfehler*) erzeugt.
- Eine solche Exception wird mittels der sogenannten *Exception Guard* erkannt: Für jede Prozedur p mit formalen Parametern x_1, \dots, x_n ist $except_p [x_1, \dots, x_n]$ ein boolscher Term mit

$except_p [t_1, \dots, t_n]$ gilt *gdw.* die Ausführung von $p(t_1, \dots, t_n)$ (ohne rekursive Aufrufe) im Prozedurrumpf von p zu \star führt.

- *Beispiel:* $except_{last} [k] = ?\emptyset(k)$
- *Beispiel:* $except_{\log_2} [x] = \text{if}\{?0(x), \text{true}, \text{if}\{?0(\neg(x)), \text{false}, \neg\text{even}(x)\}\}$
- *Beispiel:* $except_p [x] = \text{false}$ für alle Prozeduren p ohne \star im Prozedurrumpf

- Anzeige in *VeriFun: Programm Viewer* \ Termination \ Exception Guard



2 Operationale Semantik von \mathcal{L}

- Mit Beweisen im *HPL*-Kalkül sollen Behauptungen über \mathcal{L} -Programme bewiesen werden
- **Konsequenz:** Der *HPL*-Kalkül muß die *Berechnungen* eines \mathcal{L} -Programms reflektieren
- Die *Berechnungen* eines \mathcal{L} -Programms werden durch Angabe eines *Berechnungskalküls* definiert
- Der *Berechnungskalkül* setzt die Norm für die Implementierung der Programmiersprache \mathcal{L}
- **Damit:** Durch den *Berechnungskalkül* wird eine *operationale Semantik* für \mathcal{L} definiert:
 - “Semantik”: heißt “Bedeutung”
 - “operational”: *konkrete Angabe*, wie ein *Interpreter* Ausdrücke der Sprache *ausrechnet*

2.1 Der Berechnungskalkül

- Ist jeweils für ein konkretes \mathcal{L} -Programm P definiert
- **Sprache:** *Grundterme* (= Terme ohne Variable) über den Funktionssymbolen, die durch die Datentyp- und Prozedurdefinitionen gegeben sind
- **Berechnungsregeln:** Ausdrücke der Form

$$\frac{t}{r}, \text{ falls } \mathcal{B}(t, r)$$

mit Bedeutung “ersetze Grundterm t durch Grundterm r , falls die Bedingung $\mathcal{B}(t, r)$ auf t und r zutrifft”

- **Notation:**
 - $\mathcal{G}(P)$ bezeichnet alle *Grundterme* von P , $\mathcal{C}(P)$ bezeichnet alle *Konstruktorgrundterme* von P (damit: $\mathcal{C}(P) \subsetneq \mathcal{G}(P)$)
 - für $t, t' \in \mathcal{G}(P)$ schreiben wir $t \Rightarrow_P t'$ gdw. t' aus t durch Anwendung einer Berechnungsregel entsteht
 - \Rightarrow_P^+ ist *transitive Hülle* von \Rightarrow_P und \Rightarrow_P^* ist *reflexive Hülle* von \Rightarrow_P^+ .
 - für $t, t' \in \mathcal{G}(P)$ gilt $t \Rightarrow_P^! t'$ gdw. $t \Rightarrow_P^* t'$ für ein $t' \in \mathcal{G}(P)$ und $t' \not\Rightarrow_P t''$ für alle $t'' \in \mathcal{G}(P)$.
 - für $t \in \mathcal{G}(P)$ ist $t_{\downarrow P}$ definiert als t' gdw. $t \Rightarrow_P^! t'$ für genau ein $t' \in \mathcal{G}(P)$.

Berechnungsregeln für Konstruktoren, Selektoren und Gleichheit

$$\frac{?cons_i(cons_i(q_1, \dots, q_{n_i}))}{true}, \text{ falls } q_1, \dots, q_{n_i} \in \mathcal{C}(P) \quad (3)$$

$$\frac{?cons_j(cons_i(q_1, \dots, q_{n_i}))}{false}, \text{ falls } q_1, \dots, q_{n_i} \in \mathcal{C}(P) \text{ und } j \neq i \quad (4)$$

$$\frac{sel_{i,h}(cons_i(q_1, \dots, q_{n_i}))}{q_h}, \text{ falls } q_1, \dots, q_{n_i} \in \mathcal{C}(P) \quad (5)$$

$$\frac{eq(q_1, q_2)}{true}, \text{ falls } q_1, q_2 \in \mathcal{C}(P) \text{ und } q_1 = q_2 \quad (6)$$

$$\frac{eq(q_1, q_2)}{false}, \text{ falls } q_1, q_2 \in \mathcal{C}(P) \text{ und } q_1 \neq q_2 \quad (7)$$

Berechnungsregeln für bedingte Ausdrücke

$$\frac{\textit{if}\{b, t_1, t_2\}}{\textit{if}\{b', t_1, t_2\}}, \text{ falls } b \Rightarrow_P b' \quad (8)$$

$$\frac{\textit{if}\{\textit{true}, t_1, t_2\}}{t_1} \quad (9)$$

$$\frac{\textit{if}\{\textit{false}, t_1, t_2\}}{t_2} \quad (10)$$

$$\frac{\textit{case}\{c; \textit{cons}_1 : t_1, \dots, \textit{cons}_k : t_k\}}{\textit{case}\{c'; \textit{cons}_1 : t_1, \dots, \textit{cons}_k : t_k\}}, \text{ falls } c \Rightarrow_P c' \quad (11)$$

$$\frac{\textit{case}\{\textit{cons}_i(q_1, \dots, q_{n_i}); \textit{cons}_1 : t_1, \dots, \textit{cons}_k : t_k\}}{t_i}, \text{ falls } q_1, \dots, q_{n_i} \in \mathcal{C}(P) \quad (12)$$

$$\frac{\text{case}\{c; \text{cons}_{\pi(1)} : t_{\pi(1)}, \dots, \text{cons}_{\pi(h)} : t_{\pi(h)}, \text{other} : t\}}{\text{case}\{c'; \text{cons}_{\pi(1)} : t_{\pi(1)}, \dots, \text{cons}_{\pi(h)} : t_{\pi(h)}, \text{other} : t\}}, \quad (13)$$

falls $c \Rightarrow_P c'$ mit $h < k$ und π bijektiv auf $\{1, \dots, k\}$

$$\frac{\text{case}\{\text{cons}_i(q_1, \dots, q_{n_i}); \text{cons}_{\pi(1)} : t_{\pi(1)}, \dots, \text{cons}_{\pi(h)} : t_{\pi(h)}, \text{other} : t\}}{t_i},$$

falls $q_1, \dots, q_{n_i} \in \mathcal{C}(P)$ und $i = \pi(j)$ für ein $j \leq h < k$ mit π bijektiv auf $\{1, \dots, k\}$
(14)

$$\frac{\text{case}\{\text{cons}_i(q_1, \dots, q_{n_i}); \text{cons}_{\pi(1)} : t_{\pi(1)}, \dots, \text{cons}_{\pi(h)} : t_{\pi(h)}, \text{other} : t\}}{t},$$

falls $q_1, \dots, q_{n_i} \in \mathcal{C}(P)$ und $i \neq \pi(j)$ für alle $j \leq h < k$ mit π bijektiv auf $\{1, \dots, k\}$
(15)

Berechnungsregeln für *let*-Ausdrücke

$$\frac{\text{let}\{var := t; r\}}{\text{let}\{var := t'; r\}}, \text{ falls } t \Rightarrow_P t' \quad (16)$$

$$\frac{\text{let}\{var := t; r\}}{r [var/t]}, \text{ falls } t = t_{\downarrow_P} \quad (17)$$

Berechnungsregeln für Funktionsanwendungen

$$\frac{f(t_1, \dots, t_i, \dots, t_n)}{f(t_1, \dots, t'_i, \dots, t_n)}, \text{ falls } f \notin \{if, case, let\} \text{ und } t_i \Rightarrow_P t'_i \quad (18)$$

$$\frac{\text{proc}(q_1, \dots, q_k)}{\text{body}_{\text{proc}} [x_1/q_1, \dots, x_k/q_k]}, \text{ falls } \begin{array}{l} \text{proc bezeichnet Prozedur, } q_1, \dots, q_k \in \mathcal{C}(P) \\ \text{und } \text{except}_{\text{proc}} [x_1/q_1, \dots, x_k/q_k]_{\downarrow_P} = \text{false} \end{array} \quad (19)$$

2.2 Eigenschaften von \Rightarrow_P

- Jeder Konstruktorgrundterm ist \Rightarrow_P -*minimal*, d.h. $q \Rightarrow_P^! q$ für alle $q \in \mathcal{C}(P)$.

Grund: Keine Berechnungsregel ist auf ein $q \in \mathcal{C}(P)$ anwendbar.

Bedeutung: *Konstruktorgrundterme* bezeichnen *Werte*, diese kann man nicht weiter ausrechnen (genausowenig wie man 5 ausrechnen kann).

- \Rightarrow_P ist *nicht fundiert* (und damit ist $(\mathcal{G}(P), \Rightarrow_P)$ *keine fundierte Menge*).

Grund: P kann *nicht-terminierende Prozeduren* enthalten.²

- \Rightarrow_P ist *nicht deterministisch*.

Grund: Berechnungsregel (18) – die *Reihenfolge*, in der Argumente ausgerechnet werden, ist *nicht festgelegt*.

Beispiel:

$\text{insert}(\text{pred}(\text{succ}(0)), \text{tl}(0 :: \emptyset)) \Rightarrow_P \text{insert}(0, \text{tl}(0 :: \emptyset))$

$\text{insert}(\text{pred}(\text{succ}(0)), \text{tl}(0 :: \emptyset)) \Rightarrow_P \text{insert}(\text{pred}(\text{succ}(0)), \emptyset)$

Frage: Welcher Berechnungsschritt soll durchgeführt werden ?

² Fundierte Relationen und Mengen => **Kapitel 6**, Terminierung => **Kapitel 8**.

- \Rightarrow_P ist *konfluent*, d.h. für alle $t, t_1, t_2 \in \mathcal{G}(P)$ mit $t_1 \stackrel{*}{\leftarrow}_P t \Rightarrow_P^* t_2$ existiert ein $r \in \mathcal{G}(P)$ mit $t_1 \Rightarrow_P^* r \stackrel{*}{\leftarrow}_P t_2$.

Grund: Mit Regel (18) können *alle* Argumente ausgerechnet werden.

Beispiel:

- $\text{insert}(0, \text{tl}(0 :: \emptyset)) \Rightarrow_P \text{insert}(0, \emptyset)$
- $\text{insert}(\text{pred}(\text{succ}(0)), \emptyset) \Rightarrow_P \text{insert}(0, \emptyset)$

Damit gilt (1):

Wenn $t \Rightarrow_P^! r$ **für ein** $r \in \mathcal{G}(P)$, dann $r = r'$ **für alle** $r' \in \mathcal{G}(P)$ mit $t \Rightarrow_P^! r'$

Also: Wenn $t \Rightarrow_P^! r$ für ein $r \in \mathcal{G}(P)$, dann $t \downarrow_P = r$

Bedeutet: Wenn es *eine erfolgreiche* (= terminierende) Berechnung von t gibt, so liefern *alle erfolgreichen* Berechnungen von t das *gleiche Ergebnis*

Weiter gilt (2): Wenn $t \Rightarrow_P^* t' \not\Rightarrow_P^! r$ für ein $t' \in \mathcal{G}(P)$ und alle $r \in \mathcal{G}(P)$, dann $t \not\Rightarrow_P^! r'$ für alle $r' \in \mathcal{G}(P)$

Bedeutet: Wenn es *eine erfolglose* (= nicht terminierende) Berechnung von t gibt, so sind *alle* Berechnungen von t *erfolglos*

Konsequenz aus (1) und (2):

Der Indeterminismus von \Rightarrow_P kann *beliebig aufgelöst werden* !

Bemerkung 5 (*Parameterübergabe “call-by-value”*)

Mit den Berechnungsregeln (18) und (19) werden Prozeduraufrufe call-by-value ausgeführt, d.h. Prozeduren werden nur mit ausgerechneten aktuellen Parametern ausgeführt.

2.3 Der Interpreter $eval_P$

- Wegen der Konfluenz von \Rightarrow_P existiert für jeden Grundterm $t \in \mathcal{G}(P)$ höchstens ein Grundterm $r \in \mathcal{G}(P)$ mit $t \Rightarrow_P^! r$, also $r = t_{\Downarrow_P}$ (falls solch ein r existiert).
- Damit ist $eval_P : \mathcal{G}(P) \mapsto \mathcal{G}(P)$ gegeben durch

$$eval_P(t) := \begin{cases} t_{\Downarrow_P} & , \text{ falls } t \Rightarrow_P^! r \text{ für ein } r \in \mathcal{G}(P) \\ \text{undefiniert} & , \text{ sonst.} \end{cases}$$

wohldefiniert.

- $eval_P$ ist der *Interpreter* für ein \mathcal{L} -Programm P
- Mit $eval_P$ werden *Grundterme* des \mathcal{L} -Programms P *ausgerechnet*.

Beispiel 3 (*Erfolglose Berechnung von \mathcal{L} -Ausdrücken*)

- Für

```
function foo(x:nat):nat <= succ(foo(x)) end
```

- gilt $eval_P(\text{foo}(0)) = \text{undefiniert}$,
- denn $\text{foo}(0) \not\Rightarrow_P^! r$ für alle $r \in \mathcal{G}(P)$:

$$\begin{aligned} & \underline{\text{foo}(0)} \\ \Rightarrow_P & \text{succ}(\underline{\text{foo}(0)}) && \text{, mit (19)} \\ \Rightarrow_P & \text{succ}(\text{succ}(\underline{\text{foo}(0)})) && \text{, mit (18) und (19)} \\ \Rightarrow_P & \text{succ}(\text{succ}(\text{succ}(\underline{\text{foo}(0)}))) && \text{, mit (18) und (19)} \\ \Rightarrow_P & \dots && \text{, mit (18) und (19)} \end{aligned}$$

Beispiel 4 (*Erfolgreiche Berechnung von \mathcal{L} -Ausdrücken*)

- Für

```
function plus(x : nat, y : nat) : nat <=
  if ?0(x)
    then y
    else succ(plus(pred(x), y))
end_if
```

- gilt $eval_P(\text{plus}(\text{succ}(\text{succ}(0)), \text{succ}(\text{succ}(0))))$
 $= \text{succ}(\text{succ}(\text{succ}(\text{succ}(0))))$,
- also $eval_P(2 + 2) = 4$,
- denn $\text{plus}(\text{succ}(\text{succ}(0)), \text{succ}(\text{succ}(0)))$
 $\Rightarrow_P^! \text{succ}(\text{succ}(\text{succ}(\text{succ}(0))))$

wegen ...

plus (succ (succ (0)), succ (succ (0)))

\Rightarrow_P if { ?0 (succ (succ (0))),
 succ (succ (0)), , mit (19)
 succ (plus (pred (succ (succ (0))),
 succ (succ (0)))) }

\Rightarrow_P if { false,
succ (succ (0)), , mit (8) und (4)
succ (plus (pred (succ (succ (0))),
succ (succ (0)))) }

\Rightarrow_P succ (plus (pred (succ (succ (0))),
 succ (succ (0)))) , mit (10)

\Rightarrow_P succ (plus (succ (0),
succ (succ (0)))) , mit (18) und (5)

$$\Rightarrow_P \text{succ}(\text{if}\{\underline{?0(\text{succ}(0))}, \text{succ}(\text{succ}(0)), \text{succ}(\text{plus}(\text{pred}(\text{succ}(0)), \text{succ}(\text{succ}(0))))\}) \quad , \text{ mit (18) und (19)}$$

$$\Rightarrow_P \text{succ}(\text{if}\{\underline{\text{false}}, \underline{\text{succ}(\text{succ}(0))}, \underline{\text{succ}(\text{plus}(\text{pred}(\text{succ}(0)), \text{succ}(\text{succ}(0))))}\}) \quad , \text{ mit (18), (8) und (4)}$$

$$\Rightarrow_P \text{succ}(\text{succ}(\text{plus}(\underline{\text{pred}(\text{succ}(0))}, \text{succ}(\text{succ}(0)))) \quad , \text{ mit (18) und (10)}$$

$$\Rightarrow_P \text{succ}(\text{succ}(\underline{\text{plus}(0, \text{succ}(\text{succ}(0)))}) \quad , \text{ mit (18) und (5)}$$

$$\begin{aligned} & \text{succ}(\text{succ}(\text{if}\{\underline{?0(0)}, \\ \Rightarrow_P & \quad \text{succ}(\text{succ}(0)), \quad , \text{ mit (18) und (19)} \\ & \quad \text{succ}(\text{plus}(\text{pred}(0), \\ & \quad \quad \text{succ}(\text{succ}(0))))\})) \end{aligned}$$

$$\begin{aligned} & \text{succ}(\text{succ}(\underline{\text{if}\{\text{true}, \\ \Rightarrow_P & \quad \underline{\text{succ}(\text{succ}(0))}, \quad , \text{ mit (18), (8) \& (3)} \\ & \quad \underline{\text{succ}(\text{plus}(\text{pred}(0), \\ & \quad \quad \underline{\text{succ}(\text{succ}(0))))\})) \end{aligned}$$

$$\Rightarrow_P \text{succ}(\text{succ}(\text{succ}(\text{succ}(0)))) \quad , \text{ mit (18) und (9)}$$

- **Ebenso:**

$$\text{eval}_P(\text{isort}(2 :: 4 :: 3 :: 1 :: \emptyset)) = 1 :: 2 :: 3 :: 4 :: \emptyset,$$

denn

$$\text{isort}(2 :: 4 :: 3 :: 1 :: \emptyset) \Rightarrow_P^! 1 :: 2 :: 3 :: 4 :: \emptyset$$

2.4 Stuck-Terme

- Ein Term $t \in \mathcal{G}(P) \setminus \mathcal{C}(P)$ mit $t \Rightarrow_P^! t$ wird *Stuck-Term* genannt.
- **Bedeutet:** *Stuck-Terme* können wie Konstruktorgrundterme (also Terme aus $\mathcal{C}(P)$) nicht weiter ausgerechnet werden, sind aber – im Unterschied zu Termen aus $\mathcal{C}(P)$ – *keine* “Werte”.
- *Stuck-Terme* entstehen durch
 - (a) Anwendungen von *Selektoren* auf *Konstruktoren*, zu denen sie *nicht gehören*. **Beispiele:**
 - * $-(0)$, $\text{hd}(\emptyset)$, $\text{car}(\text{nil})$, $\text{data}(\text{cons}(\dots))$, $\text{cdr}(\text{atom}(\dots))$
(für die Datentypen `nat`, `list` und `sexpr` aus **Kapitel 2**)
 - (b) *Prozeduraufrufe* mit aktuellen Parametern, für die die *Exception Guard nicht* zu `false` ausgerechnet wird, vgl. Regel (19). **Beispiele:**
 - * $\text{last}(\emptyset)$, $\log_2(0)$, $\log_2(3)$ für die Prozeduren aus Abschnitt 1.3.3
 - (c) *bedingte Ausdrücke*, deren *Bedingung* ein *Stuck-Term* ist. **Beispiele:**
 - * $\text{if}\{?0(\log_2(3)), \dots, \dots\}$, $\text{case}\{\text{car}(\text{nil}); \dots\}$
 - (d) *Anwendungen* von Funktionssymbolen $\notin \{\text{if}, \text{case}\}$ auf *Stuck-Terme*. **Beispiele:**
 - * $-(-(0))$, $+(-(0))$, $\text{cdr}(\text{car}(\text{nil}))$, $\text{cons}(\text{nil}, \text{cdr}(\text{nil}))$,
 $\text{insert}(\text{last}(\emptyset), \emptyset)$.

Also: Die *Laufzeitfehler* (= *Exceptions*), die bei Ausrechnen eines Terms durch einen *konkreten Interpreter* entstehen (vgl. Abschnitt 1.3.3), werden in der *operationalen Semantik* von \mathcal{L} durch *Stuck-Terme* **modelliert**.

- **Beispiel** (für Prozedur \log_2 aus Abschnitt 1.3.3):

$$\begin{array}{ll}
 - \log_2(0) \Rightarrow_P^! \mathbf{log_2(0)} & - \log_2(4) \Rightarrow_P^! 2 \\
 - \log_2(1) \Rightarrow_P^! 0 & - \log_2(5) \Rightarrow_P^! \mathbf{log_2(5)} \\
 - \log_2(2) \Rightarrow_P^! 1 & - \log_2(6) \Rightarrow_P^! \mathbf{^+(\log_2(3))} \\
 - \log_2(3) \Rightarrow_P^! \mathbf{log_2(3)} & - \log_2(7) \Rightarrow_P^! \mathbf{log_2(7)}
 \end{array}$$

Unterschied Interpreter $eval_P$ / Symbolischer Interpreter (Symbolic Evaluator):

- Der symbolische Interpreter behandelt *Stuck-Terme* wie “normale” Terme

- **Beispiele:**

$$\begin{array}{l}
 - eval_P(\text{^-}(0) = \text{^-}(0)) = \text{^-}(0) = \text{^-}(0) \\
 - sym\text{-}eval_P(\text{^-}(0) = \text{^-}(0)) = \text{true} \\
 - eval_P(\text{if}\{\text{^-}(0) = \text{^-}(0), 1, 2\}) = \text{if}\{\text{^-}(0) = \text{^-}(0), 1, 2\} \\
 - sym\text{-}eval_P(\text{if}\{\text{^-}(0) = \text{^-}(0), 1, 2\}) = 1
 \end{array}$$