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:

$$\begin{aligned}&\textbf{structure}\ \texttt{struc}[@\texttt{T}_1,\ldots,@\texttt{T}_n]\ \texttt{<=}\\ &\quad\texttt{cons}_1\texttt{(sel}_{1,1}\texttt{: struc}_{1,1},\ldots,\texttt{sel}_{1,n_1}\texttt{: struc}_{1,n_1}\texttt{)},\\ &\quad\vdots\\ &\quad\texttt{cons}_k\texttt{(sel}_{k,1}\texttt{: struc}_{k,1},\ldots,\texttt{sel}_{k,n_k}\texttt{: struc}_{k,n_k}\texttt{)}\end{aligned} \tag{1}$$

mit

(1) $@\texttt{T}_1,\ldots,@\texttt{T}_n$ 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 $i \in \{1,\ldots,k\}$, d.h. Konstruktor*konstante* sind erlaubt,

(4) für alle $i \in \{1,\ldots,k\}$ und alle $h \in \{1,\ldots,n_i\}$ ist $\texttt{struc}_{i,h}$ ein Datentyp mit

  (a) $\texttt{struc}_{i,h}$ ist *Instanz* eines *bereits definierten* Datentyps oder es gilt $\texttt{struc}_{i,h} = \texttt{struc}[@\texttt{T}_1,\ldots,@\texttt{T}_n]$ (= *rekursive Definition erlaubt*)

  (b) $\texttt{struc}_{i,h} \neq \texttt{bool}$,

  (c) in $\texttt{struc}_{i,h}$ werden höchstens die Typvariablen aus $\{@\texttt{T}_1,\ldots,@\texttt{T}_n\}$ verwendet,

  (d) die Bezeichner $\texttt{struc}$, $\texttt{cons}_i$, $\texttt{sel}_{i,h}$ sind voneinander verschieden und bislang nicht verwendet worden

(5) für *ein* $i \in \{1,\ldots,k\}$ und alle $h \in \{1,\ldots,n_i\}$ ist $\texttt{struc}_{i,h}$ *verschieden von* $\texttt{struc}[@\texttt{T}_1,\ldots,@\texttt{T}_n]$. ■

Dabei gilt:

Definition 2 *(Instanzen von Datentypen)*
Ein Datentyp $\texttt{struc}'$ *ist eine* Instanz *eines Datentyps* `struc` *gdw.* $\texttt{struc}'$ *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 $\mathtt{eq}_{struc}$ mit Signatur
  $\mathtt{eq}_{struc} : \mathtt{struc}[@\mathtt{T}_1, \ldots, @\mathtt{T}_n] \times \mathtt{struc}[@\mathtt{T}_1, \ldots, @\mathtt{T}_n] \to \mathtt{bool}$ und
- ein Funktionssymbol $\mathtt{if}_{struc}$ mit Signatur
  $\mathtt{if}_{struc} : \mathtt{bool} \times \mathtt{struc}[@\mathtt{T}_1, \ldots, @\mathtt{T}_n] \times \mathtt{struc}[@\mathtt{T}_1, \ldots, @\mathtt{T}_n] \to \mathtt{struc}[@\mathtt{T}_1, \ldots, @\mathtt{T}_n]$

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

- ein “neues” *Konstruktor*funktionssymbol $\mathtt{cons}_i$ mit Signatur
  $\mathtt{cons}_i : \mathtt{struc}_{i,1} \times \ldots \times \mathtt{struc}_{i,n_i} \to \mathtt{struc}[@\mathtt{T}_1, \ldots, @\mathtt{T}_n]$
- ein “neues” *Strukturprädikat*ssymbol $\mathtt{?cons}_i$ mit Signatur
  $\mathtt{?cons}_i : \mathtt{struc}[@\mathtt{T}_1, \ldots, @\mathtt{T}_n] \to \mathtt{bool}$
- für jede Argumentposition $h \in \{1, \ldots, n_i\}$ eines “neuen” Konstruktors $\mathtt{cons}_i$ ein “neues” *Selektor*funktionssymbol $\mathtt{sel}_{i,h}$ mit Signatur
  $\mathtt{sel}_{i,h} : \mathtt{struc}[@\mathtt{T}_1, \ldots, @\mathtt{T}_n] \to \mathtt{struc}_{i,h}$

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**:
  - “$\mathtt{eq}_{struc}(\mathtt{x}, \mathtt{y})$” wird im PrettyPrint von *VeriFun* als “`x = y`” angezeigt
  - Bei Eingaben *muß* “`x = y`” anstatt “$\mathtt{eq}_{struc}(\mathtt{x}, \mathtt{y})$” geschrieben werden
  - Um Verwechselungen mit “$=$” der Metasprache zu vermeiden wird “$\mathtt{eq}_{struc}$” jedoch außerhalb von Beispielen und Screenshots verwendet

Bemerkung 3 *(Strukturprädikate)*

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

1.2 Prozeduren in $\mathcal{L}$

Definition 3 (Allgemeine Form einer Prozedurdefinition)
Prozeduren werden in $\mathcal{L}$ definiert durch Ausdrücke der Form:

$$\textbf{function}\ \mathtt{proc}(\mathtt{x}_1\mathtt{:struc}_1, \ldots, \mathtt{x}_k\mathtt{:struc}_k)\mathtt{:struc} <= body_{\mathtt{proc}} \qquad (2)$$

mit

(1) $k \geq 1$, d.h. *Konstantensymbole* sind immer *Konstruktorsymbole*,

(2) für alle $i \in \{1, \ldots, k\}$ ist $\mathtt{struc}_i$ Instanz eines bereits definierten Datentyps $\neq$ `bool`

(3) der Bezeichner `proc` ist bislang nicht verwendet worden

(4) `struc` ist Instanz eines bereits definierten Datentyps in dem höchstens Typvariable aus den $\mathtt{struc}_i$ vorkommen

(5) $body_{\mathtt{proc}}$ (= *Prozedurrumpf*) ist ein Term vom Typ `struc` über

- den paarweise verschiedenen Variablensymbolen $x_1, \ldots, x_k$ (= *formale Parameter*) mit “Datentyp von x_i ist $\mathtt{struc}_i$” und
- den durch bereits definierte Datentypen und Prozeduren eingeführten Funktionssymbolen erweitert um `proc` (= *rekursive Definition möglich*). ■

Bedeutung:

Ein Ausdruck der Form (2) definiert ein "neues" Prozedurfunktionssymbol `proc` mit Signatur $\texttt{proc:struc}_1 \times \ldots \times \texttt{struc}_k \to \texttt{struc}$

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 $r_1, \ldots, r_n$ eines Typs $\texttt{struc}'$ erhält man eine strukturelle Fallunterscheidung bzgl. `struc` durch

$$\begin{array}{l} \texttt{case}\ t\ \texttt{of} \\ \quad cons_1 : r_1, \\ \quad cons_2 : r_2, \\ \quad\quad \ldots \\ \quad cons_k : r_k \\ \texttt{end\_case} \end{array} \quad \text{bzw.} \quad \begin{array}{l} \texttt{case}\{t\ ; \\ \quad cons_1 : r_1, \\ \quad cons_2 : r_2, \\ \quad\quad \ldots \\ \quad cons_k : r_k\} \end{array}$$

(Prozedurale Schreibweise) *(Funktionale Schreibweise)*

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

$$\begin{array}{l} \texttt{case}\ t\ \texttt{of} \\ \quad cons_i : r_i, \\ \quad cons_j : r_j, \\ \quad\quad \ldots \\ \quad \texttt{other} : r \\ \texttt{end\_case} \end{array} \quad \text{bzw.} \quad \begin{array}{l} \texttt{case}\{t\ ; \\ \quad cons_i : r_i, \\ \quad cons_j : r_j, \\ \quad\quad \ldots \\ \quad \texttt{other} : r\} \end{array}$$

(Prozedurale Schreibweise) *(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 ?ø(k)
 then 0
 else if ?ø(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

  `let` var `:=` t `in` r `end_let` *bzw.* `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
  (=> erhöht zuweilen die Lesbarkeit von Zieltermen im *Proof Viewer*)

1.3.3 Partiell definierte Prozeduren

Für Prozeduren mit *monomorphem* Ergebnistyp können wir für *Argumente außerhalb des Definitionsbereichs* der zu berechnenden Funktion ein *willkürlich gewähltes Ergebnis* angeben[1], etwa

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

Aber:

- Willkürliches und unintuitives Ergebnis für `minimum(ø)` !
- Was machen wir bei *polymorphen* Ergebnistypen ?

[1] Natürlich nur dann, wenn der *Definitionsbereich* der zu berechnenden Funktion *entscheidbar* ist (=> Vorlesung "Berechenbarkeit").

Beispiel:

```
function last(k : list[@T]) : @T <=
if ?ø(k)
 then ???
 else if ?ø(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 `last(ø)` ?

Problem: Wegen *Polymorphie* kann für `last(ø)` *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` $\star$ `end_if` **statt** `if` $cond$ `then` $term$ `end_if`
  - `if` $cond$ `then` $\star$ `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 ?ø(k)
 then ⋆
 else if ?ø(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 $n = 2^m$.

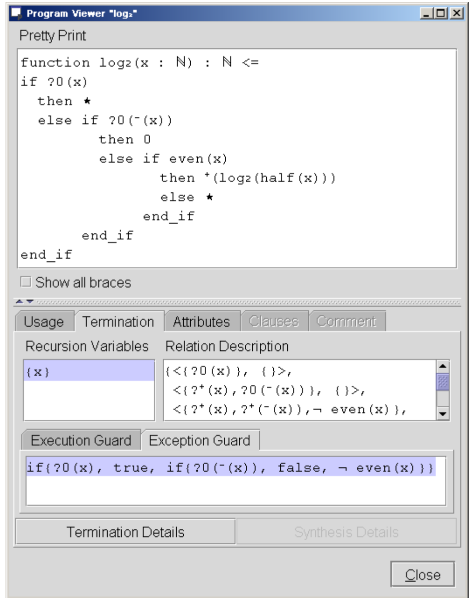
Die Exception Guard

- **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 $x_1, \ldots, x_n$ ist $except_{\mathtt{p}}[x_1, \ldots, x_n]$ ein boolscher Term mit

> $except_{\mathtt{p}}[t_1, \ldots, t_n]$ gilt *gdw.* die Ausführung von $\mathtt{p}(t_1, \ldots, t_n)$ (ohne rekursive Aufrufe) im Prozedurrumpf von p zu $\star$ führt.

- *Beispiel*: $except_{\mathtt{last}}[k] = \mathtt{?ø(k)}$
- *Beispiel*: $except_{\mathtt{log_2}}[x] = \mathtt{if\{?0(x),true,if\{?0(^-(x)),false,\neg even(x)\}\}}$
- *Beispiel*: $except_{\mathtt{p}}[x] = \mathtt{false}$ 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

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

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

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

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

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

Berechnungsregeln für bedingte Ausdrücke

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

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

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

$$\frac{case\{c;\ cons_1:t_1,\ldots,cons_k:t_k\}}{case\{c';\ cons_1:t_1,\ldots,cons_k:t_k\}}\text{, falls } c\Rightarrow_P c' \quad (11)$$

$$\frac{case\{cons_i(q_1,\ldots,q_{n_i});\ cons_1:t_1,\ldots,cons_k:t_k\}}{t_i}\text{, falls } q_1,\ldots,q_{n_i}\in\mathcal{C}(P) \quad (12)$$

$$\frac{case\{c;\ cons_{\pi(1)}:t_{\pi(1)},\ldots,cons_{\pi(h)}:t_{\pi(h)},other:t\}}{case\{c';\ cons_{\pi(1)}:t_{\pi(1)},\ldots,cons_{\pi(h)}:t_{\pi(h)},other:t\}}, \quad (13)$$

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

$$\frac{case\{cons_i(q_1,\ldots,q_{n_i});\ cons_{\pi(1)}:t_{\pi(1)},\ldots,cons_{\pi(h)}:t_{\pi(h)},other:t\}}{t_i},$$

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

$$\frac{case\{cons_i(q_1,\ldots,q_{n_i});\ cons_{\pi(1)}:t_{\pi(1)},\ldots,cons_{\pi(h)}:t_{\pi(h)},other:t\}}{t},$$

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

Berechnungsregeln für *let*-Ausdrücke

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

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

Berechnungsregeln für Funktionsanwendungen

$$\frac{f(t_1,\ldots,t_i,\ldots,t_n)}{f(t_1,\ldots,t_i',\ldots,t_n)}\text{, falls } f\notin\{if,case,let\} \text{ und } t_i\Rightarrow_P t_i' \quad (18)$$

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

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:
  `insert(pred(succ(0)),tl(0 :: ø ))` $\Rightarrow_P$ `insert(0, tl(0 :: ø))`
  `insert(pred(succ(0)),tl(0 :: ø ))` $\Rightarrow_P$ `insert(pred(succ(0)),ø)`

  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 $t_1 \;{}^*_P\!\Leftarrow t \Rightarrow^*_P t_2$ existiert ein $r \in \mathcal{G}(P)$ mit $t_1 \Rightarrow^*_P r \;{}^*_P\!\Leftarrow t_2$.
  Grund*:* Mit Regel (18) können *alle* Argumente ausgerechnet werden.
  Beispiel:
  – `insert(0, tl(0 :: ø))` $\Rightarrow_P$ `insert(0, ø)`
  – `insert(pred(succ(0)), ø)` $\Rightarrow_P$ `insert(0, ø)`

  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

$$eval_P(t) := \begin{cases} t_{\Downarrow_P} & \text{, falls } t \Rightarrow^!_P r \text{ für ein } r \in \mathcal{G}(P) \\[2ex] \mathit{undefiniert} & \text{, sonst.} \end{cases}$$

  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 $eval_P(\texttt{foo(0)}) = \mathit{undefiniert}$,
- denn $\texttt{foo(0)} \not\Rightarrow^!_P r$ für alle $r \in \mathcal{G}(P)$ **:**

$$\begin{array}{lll} & \underline{\texttt{foo(0)}} & \\ \Rightarrow_P & \texttt{succ(}\underline{\texttt{foo(0)}}\texttt{)} & \text{, mit (19)} \\ \Rightarrow_P & \texttt{succ(succ(}\underline{\texttt{foo(0)}}\texttt{))} & \text{, mit (18) und (19)} \\ \Rightarrow_P & \texttt{succ(succ(succ(}\underline{\texttt{foo(0)}}\texttt{)))} & \text{, mit (18) und (19)} \\ \Rightarrow_P & \ldots & \text{, mit (18) und (19)} \end{array}$$

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 $eval_P(\texttt{plus(succ(succ(0)), succ(succ(0)))})$
  $= \texttt{succ(succ(succ(succ(0))))}$,
- also $eval_P(2+2) = 4$,
- denn $\texttt{plus(succ(succ(0)), succ(succ(0))}$
  $\Rightarrow^!_P \texttt{succ(succ(succ(succ(0))))}$

  wegen ...

$$\underline{\texttt{plus(succ(succ(0)), succ(succ(0))}}$$

$$\Rightarrow_P \begin{array}{l}\texttt{if\{}\underline{\texttt{?0(succ(succ(0)))}}\texttt{,}\\ \quad\texttt{succ(succ(0)),}\\ \quad\texttt{succ(plus(pred(succ(succ(0))),}\\ \qquad\texttt{succ(succ(0)))))\}}\end{array} \qquad \text{, mit (19)}$$

$$\Rightarrow_P \begin{array}{l}\underline{\texttt{if\{false,}}\\ \quad\underline{\texttt{succ(succ(0)),}}\\ \quad\underline{\texttt{succ(plus(pred(succ(succ(0))),}}\\ \qquad\underline{\texttt{succ(succ(0)))))\}}}\end{array} \qquad \text{, mit (8) und (4)}$$

$$\Rightarrow_P \begin{array}{l}\texttt{succ(plus(}\underline{\texttt{pred(succ(succ(0))),}}\\ \qquad\texttt{succ(succ(0))))}\end{array} \qquad \text{, mit (10)}$$

$$\Rightarrow_P \begin{array}{l}\texttt{succ(}\underline{\texttt{plus(succ(0),}}\\ \qquad\underline{\texttt{succ(succ(0)))}}\texttt{)}\end{array} \qquad \text{, mit (18) und (5)}$$

$$\Rightarrow_P \begin{array}{l}\texttt{succ(if\{}\underline{\texttt{?0(succ(0))}}\texttt{,}\\ \qquad\texttt{succ(succ(0)),}\\ \qquad\texttt{succ(plus(pred(succ(0)),}\\ \qquad\quad\texttt{succ(succ(0)))))\})}\end{array} \qquad \text{, mit (18) und (19)}$$

$$\Rightarrow_P \begin{array}{l}\texttt{succ(}\underline{\texttt{if\{false,}}\\ \qquad\underline{\texttt{succ(succ(0)),}}\\ \qquad\underline{\texttt{succ(plus(pred(succ(0)),}}\\ \qquad\quad\underline{\texttt{succ(succ(0)))))\})}}\end{array} \qquad \text{, mit (18), (8) und (4)}$$

$$\Rightarrow_P \begin{array}{l}\texttt{succ(succ(plus(}\underline{\texttt{pred(succ(0)),}}\\ \qquad\qquad\texttt{succ(succ(0))))))}\end{array} \qquad \text{, mit (18) und (10)}$$

$$\Rightarrow_P \begin{array}{l}\texttt{succ(succ(}\underline{\texttt{plus(0,}}\\ \qquad\qquad\underline{\texttt{succ(succ(0)))}}\texttt{))}\end{array} \qquad \text{, mit (18) und (5)}$$

$$\Rightarrow_P \begin{array}{l}\texttt{succ(succ(if\{}\underline{\texttt{?0(0)}}\texttt{,}\\ \qquad\texttt{succ(succ(0)),}\\ \qquad\texttt{succ(plus(pred(0),}\\ \qquad\qquad\texttt{succ(succ(0)))))\}))}\end{array} \qquad \text{, mit (18) und (19)}$$

$$\Rightarrow_P \begin{array}{l}\texttt{succ(succ(}\underline{\texttt{if\{true,}}\\ \qquad\underline{\texttt{succ(succ(0)),}}\\ \qquad\underline{\texttt{succ(plus(pred(0),}}\\ \qquad\qquad\underline{\texttt{succ(succ(0)))))\}}}\texttt{))}\end{array} \qquad \text{, mit (18), (8) \& (3)}$$

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

- **Ebenso:**
  $eval_P(\texttt{isort(2 :: 4 :: 3 :: 1 :: }\varnothing\texttt{)}) = \texttt{1 :: 2 :: 3 :: 4 :: }\varnothing$,

  denn

  $\texttt{isort(2 :: 4 :: 3 :: 1 :: }\varnothing\texttt{)} \Rightarrow^!_P \texttt{1 :: 2 :: 3 :: 4 :: }\varnothing$

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), hd(ø), car(nil), data(cons(...)), cdr(atom(...))
      (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**:
    * last(ø), $\texttt{log}_2$(0), $\texttt{log}_2$(3) für die Prozeduren aus Abschnitt 1.3.3

  (c) *bedingte Ausdrücke*, deren *Bedingung* ein *Stuck-Term* ist. **Beispiele**:
    * if{?0($\texttt{log}_2$(3)), ..., ...}, case{car(nil); ...}

  (d) *Anwendungen* von Funktionssymbolen $\notin$ {if, case} auf *Stuck-Terme*. **Beispiele**:
    * $^-$($^-$(0)), $^+$($^-$(0)), cdr(car(nil)), cons(nil, cdr(nil)), insert(last(ø),ø).

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 $\texttt{log}_2$ aus Abschnitt 1.3.3):
  - $\texttt{log}_2(0) \Rightarrow^!_P \mathbf{log_2(0)}$
  - $\texttt{log}_2(1) \Rightarrow^!_P 0$
  - $\texttt{log}_2(2) \Rightarrow^!_P 1$
  - $\texttt{log}_2(3) \Rightarrow^!_P \mathbf{log_2(3)}$
  - $\texttt{log}_2(4) \Rightarrow^!_P 2$
  - $\texttt{log}_2(5) \Rightarrow^!_P \mathbf{log_2(5)}$
  - $\texttt{log}_2(6) \Rightarrow^!_P {}^+(\mathbf{log_2(3)})$
  - $\texttt{log}_2(7) \Rightarrow^!_P \mathbf{log_2(7)}$

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

- Der symbolische Interpreter behandelt *Stuck-Terme* wie "normale" Terme
- **Beispiele**:
  - $eval_P(^-(0) = {}^-(0)) = {}^-(0) = {}^-(0)$
  - $sym\text{-}eval_P(^-(0) = {}^-(0)) = \texttt{true}$
  - $eval_P(\texttt{if}\{^-(0) = {}^-(0), 1, 2\}) = \texttt{if}\{^-(0) = {}^-(0), 1, 2\}$
  - $sym\text{-}eval_P(\texttt{if}\{^-(0) = {}^-(0), 1, 2\}) = 1$