24 Strukturierte Daten

Im Folgenden wollen wir unsere funktionale Programmiersprache um einfache Datentypen erweitern. Wir beginnen mit der Einführung von Tupeln.

24.1 Tupel

Tupel werden mithilfe k-stelliger Konstruktoren ($k \geq 0$) $(.,\ldots,.)$ aufgebaut und mithilfe der Projektions-Funktionen $\#j$ ($j \in \mathbb{N}_0$) wieder zerlegt.

Entsprechend erweitern wir die Syntax unserer Programm-Ausdrücke durch:

$$\begin{array}{rcl} e & ::= & \ldots \mid (e_0,\ldots,e_{k-1}) \mid \#j\ e \\ & & \textbf{match}\ e_1\ \textbf{with}\ (x_0,\ldots,x_{k-1}) \rightarrow e_0 \end{array}$$

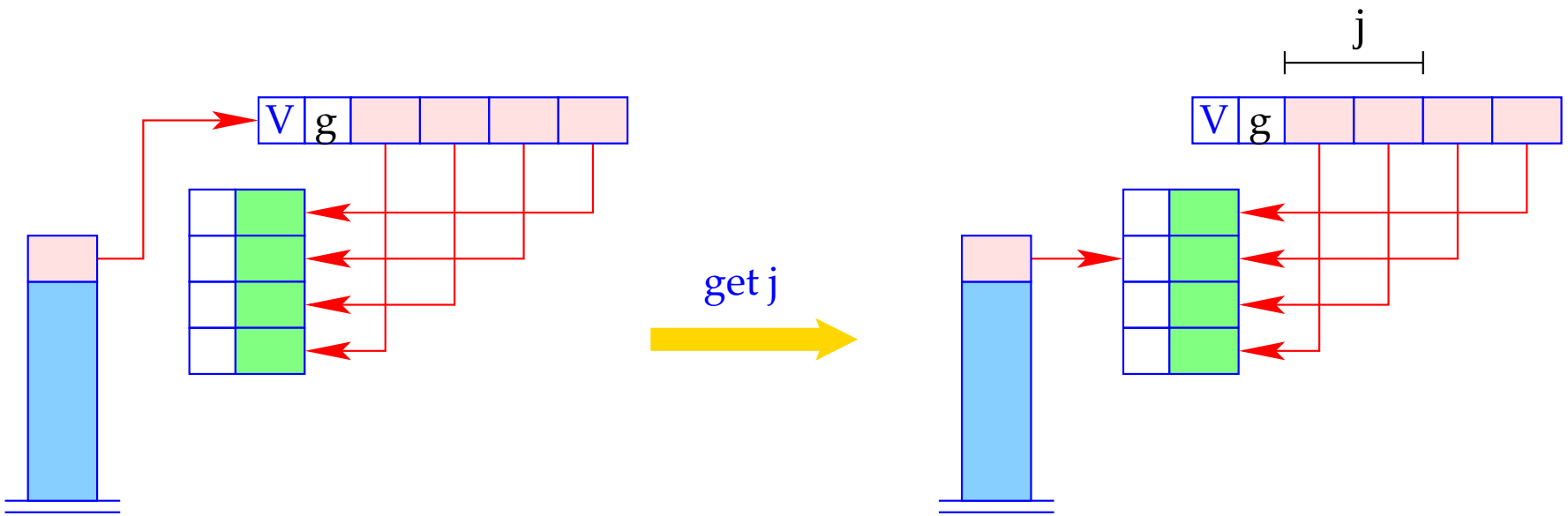
- Tupel werden angelegt, indem erst die Folge der Referenzen auf ihre Komponenten auf dem Keller gesammelt und dann mithilfe der Operation mkvec in den Heap gelegt werden.
- Auf Komponenten greifen wir zu, indem wir innerhalb des Tupels einen indizierten Zugriff vornehmen.

$$
\begin{array}{lcl}
\mathrm{code}_V\ (e_0, \ldots, e_{k-1})\ \rho\ \mathrm{kp} & = & \mathrm{code}_C\ e_0\ \rho\ \mathrm{kp} \\
 & & \mathrm{code}_C\ e_1\ \rho\ (\mathrm{kp}+1) \\
 & & \ldots \\
 & & \mathrm{code}_C\ e_{k-1}\ \rho\ (\mathrm{kp}+k-1) \\
 & & \mathrm{mkvec\ k}
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{code}_V\ (\#j\ e)\ \rho\ \mathrm{kp} & = & \mathrm{code}_V\ e\ \rho\ \mathrm{kp} \\
 & & \mathrm{get\ j} \\
 & & \mathrm{eval}
\end{array}
$$

Im Falle von CBV können natürlich direkt die Werte der e_i berechnet werden.

Dabei ist:




```
if (S[SP] == (V,g,v))
   S[SP] = v[j];
else Error "Vector expected!";
```

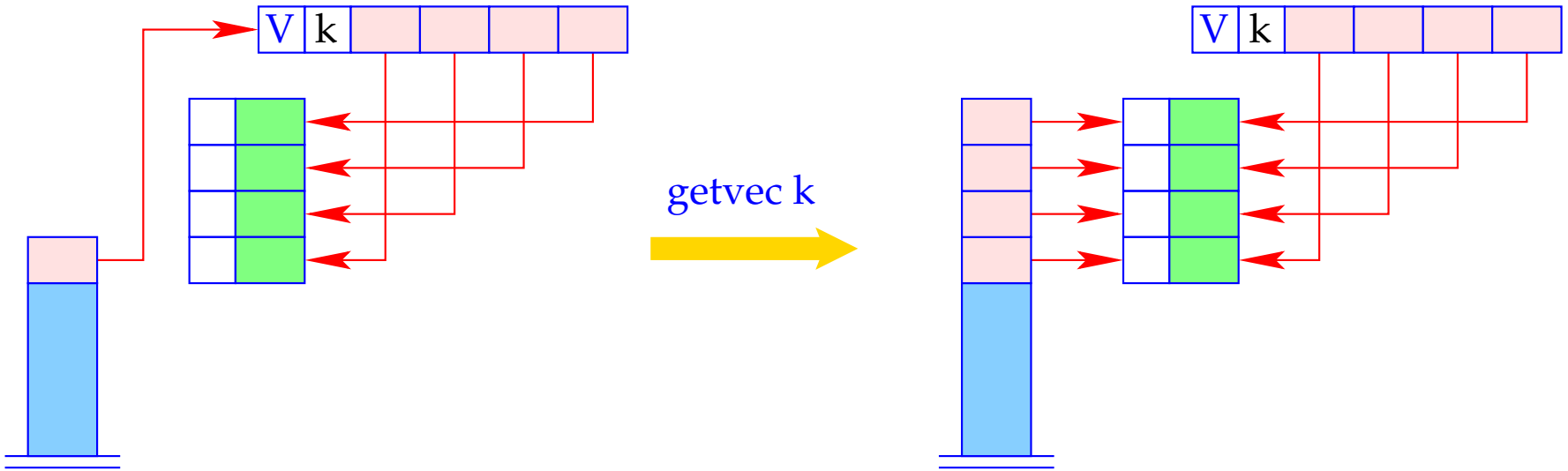
Möchte man nicht auf einzelne, sondern gegebenenfalls alle Komponenten eines Tupels zugreifen, kann man dies mithilfe des Ausdrucks
$e \equiv \textbf{match}\ e_1\ \textbf{with}\ (y_0, \ldots, y_{k-1}) \to e_0$ tun.

Diesen übersetzen wir wie folgt:

$$\begin{aligned} \text{code}_V\ e\ \rho\ \text{kp} \quad = \quad & \text{code}_V\ e_1\ \rho\ \text{kp} \\ & \text{getvec k} \\ & \text{code}_V\ e_0\ \rho'\ (\text{kp} + k) \\ & \text{slide k} \end{aligned}$$

wobei $\rho' = \rho \oplus \{y_i \mapsto (L, kp + i + 1) \mid i = 0, \ldots, k-1\}$.

Der Befehl getvec k legt die Komponenten eines Vektors der Länge k auf den Keller:




```
if (S[SP] == (V,k,v)) {
   SP--;
   for(i=0; i<k; i++) {
        SP++; S[SP] = v[i];
        }
} else Error “Vector expected!”;
```

24.2 Listen

Als Beispiel eines weiteren Datentyps betrachten wir Listen.

Listen werden aus Listen-Elementen mithilfe der Konstante [] ("Nil" – die leere Liste) und des rechts-assoziativen Operators :: ("Cons" – dem Listen-Konstruktor) aufgebaut.

Ein **match**-Ausdruck gestattet den Zugriff auf die Komponenten einer Liste.

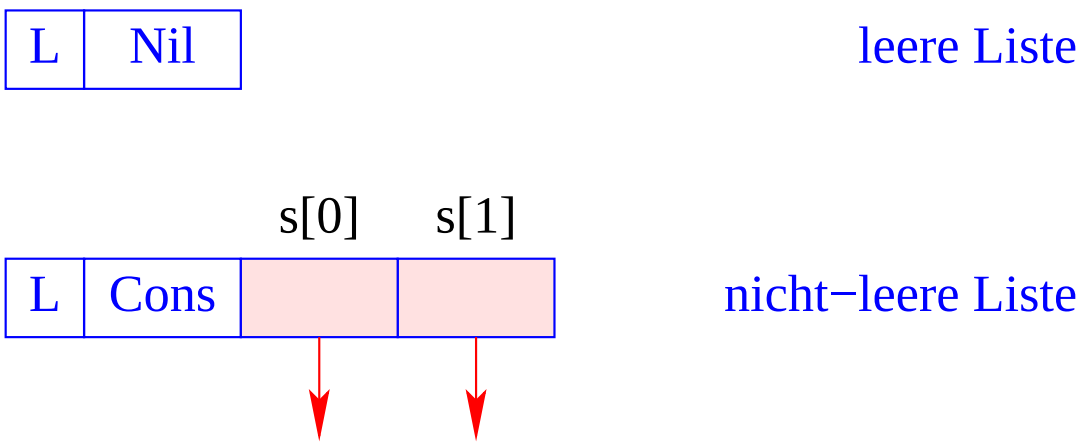
Beispiel: Die Append-Funktion app:

$$\begin{array}{lll} \textbf{let rec} \quad \text{app} \quad = & & \textbf{fun } l, y \rightarrow \textbf{match } l \textbf{ with} \\ & [] \quad \rightarrow \quad y & \\ & \mid h :: t \quad \rightarrow \quad h :: (\text{app } t\ y) & \end{array}$$

Folglich erweitern wir die Syntax von Ausdrücken e um:

$$\begin{array}{rcl} e & ::= & \ldots \mid [] \mid (e_1 :: e_2) \\ & & \mid (\textbf{match}\ e_0\ \textbf{with}\ [] \rightarrow e_1 \mid h :: t \rightarrow e_2) \end{array}$$

Neue Halden-Objekte:



24.3 Der Aufbau von Listen

Für das Anlegen von Listen-Knoten führen wir die Befehle nil und cons ein.

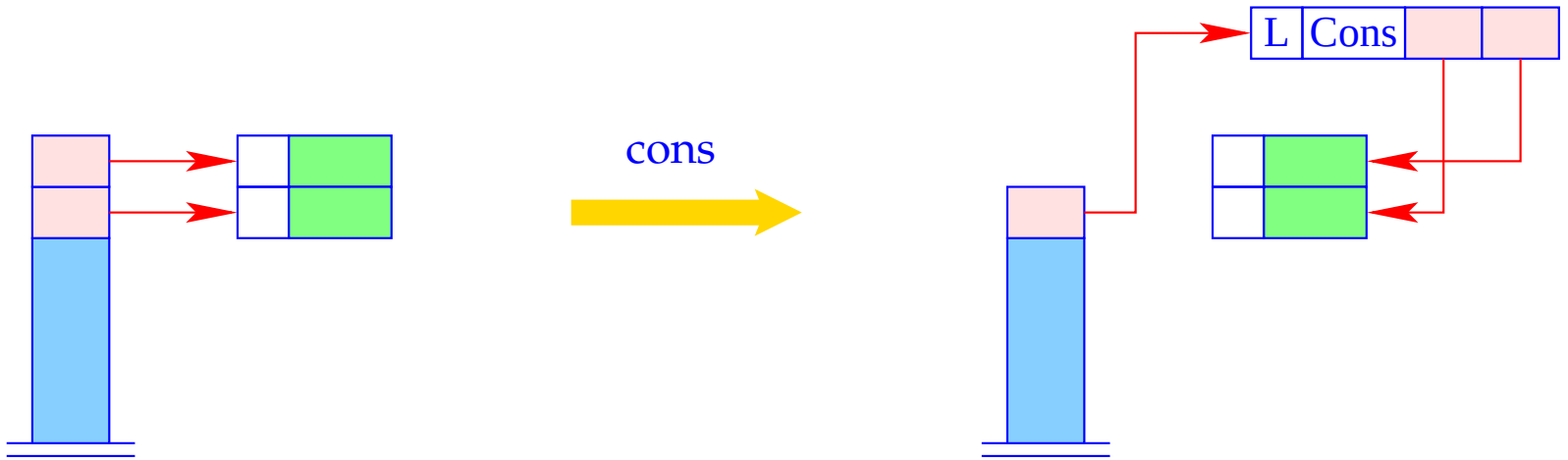
Damit erhalten wir für CBN:

$$
\begin{array}{lcl}
\text{code}_V\ [\,]\ \rho\ \text{kp} & = & \text{nil} \\
\text{code}_V\ (e_1 :: e_2)\ \rho\ \text{kp} & = & \text{code}_C\ e_1\ \rho\ \text{kp} \\
 & & \text{code}_C\ e_2\ \rho\ (\text{kp}+1) \\
 & & \text{cons}
\end{array}
$$

Beachte:

- Bei CBN werden für die Argumente von "::" Abschlüsse angelegt.
- Bei CBV müssen sie dagegen erst ausgewertet werden.

SP++; S[SP] = new (L,Nil);



S[SP-1] = new (L,Cons, S[SP-1], S[SP]);
SP- -;

24.4 Pattern-Matching

Betrachte den Ausdruck $e \equiv \textbf{match}\ e_0\ \textbf{with}\ [] \rightarrow e_1 \mid h :: t \rightarrow e_2$.

Auswertung von e erfordert:

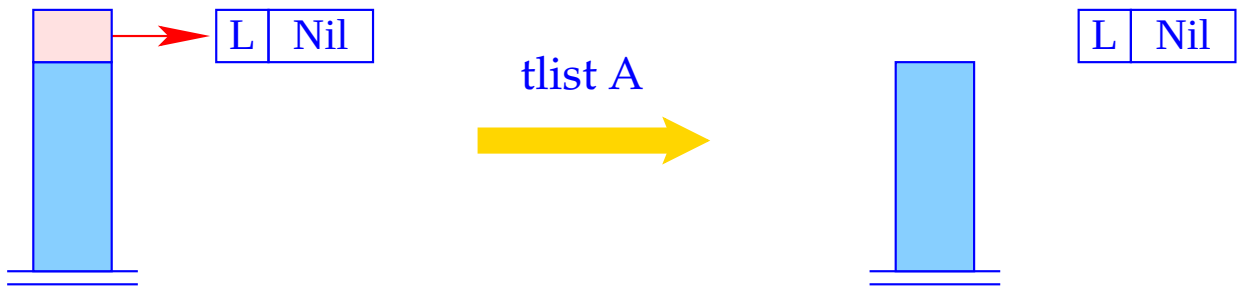
- Auswertung von e_0;
- Überprüfung, ob e_0 ein L-Objekt ist;
- Falls e_0 gleich der leeren Liste ist, Auswertung von e_1 ...
- ... andernfalls Kellern der Verweise des L-Objekts und Auswertung von e_2.

Folglich erhalten wir (für CBN wie CBV):

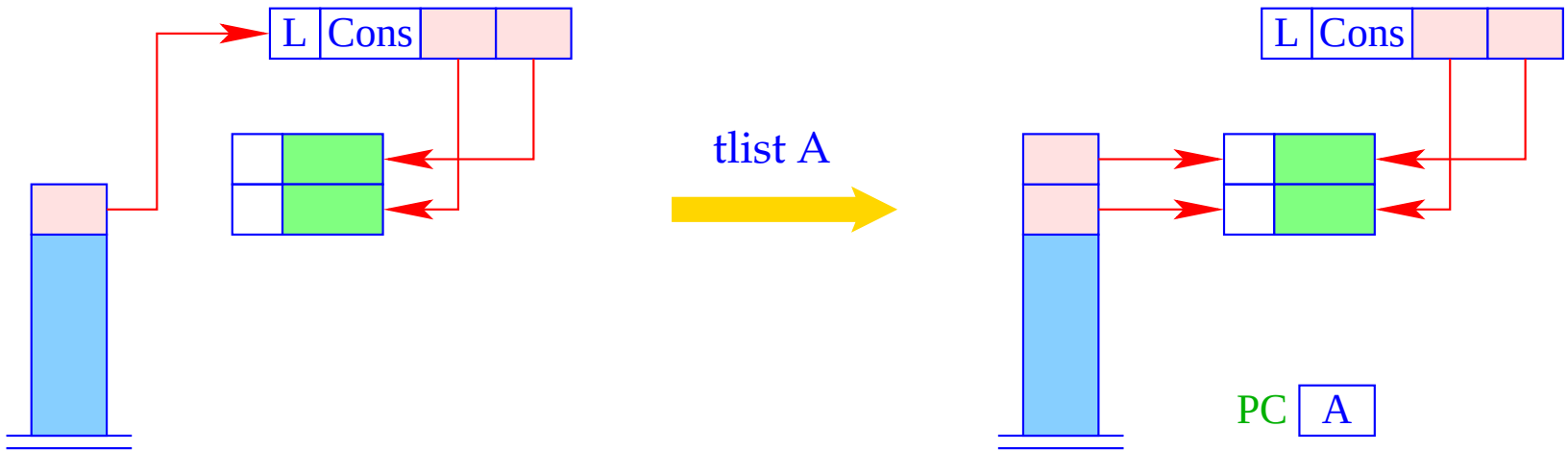
$$
\begin{array}{lll}
\text{code}_V\ e\ \rho\ \text{kp} & = & \text{code}_V\ e_0\ \rho\ \text{kp} \\
 & & \text{tlist A} \\
 & & \text{code}_V\ e_1\ \rho\ \text{kp} \\
 & & \text{jump B} \\
 & \text{A}: & \text{code}_V\ e_2\ \rho'\ (\text{kp}+2) \\
 & & \text{slide 2} \\
 & \text{B}: & \ldots
\end{array}
$$

wobei $\rho' = \rho \oplus \{h \mapsto (L, kp+1), t \mapsto (L, kp+2)\}$.

Der neue Befehl tlist A führt die notwendigen Überprüfungen durch und legt (im Cons-Fall) zwei neue lokale Variablen an:




```
h = S[SP];
if (H[h] != (L,...))
    Error “no list!”;
if (H[h] == (_,Nil)) SP- -;
     ...
```




```
... else {
   S[SP+1] = S[SP]→s[1];
   S[SP] = S[SP]→s[0];
   SP++; PC = A;
}
```

Beispiel: Der (entwirrte) Rumpf der Funktion app mit $\text{app} \mapsto (G, 0)$:

```
0        targ 2          3        pushglob 0      0    C:  mark D
0        pushloc 0       4        pushloc 2       3        pushglob 2
1        eval            5        pushloc 6       4        pushglob 1
1        tlist A         6        mkvec 3         5        pushglob 0
0        pushloc 1       4        mkclos C        6        eval
1        eval            4        cons            6        apply
1        jump B          3        slide 2         1    D:  update
2   A:   pushloc 1       1   B:   return 2
```

Beachte:

Hat man Datentypen mit mehr als zwei Konstruktoren, benötigt man eine Verallgemeinerung des tlist-Befehls, der einer `switch`-Anweisung entspricht :-)

24.5 Abschlüsse von Tupeln und Listen

Das generelle Schema für code_C lässt sich auch bei Tupeln und Listen optimieren:

$$
\begin{array}{lcll}
\text{code}_C\ (e_0,\ldots,e_{k-1})\ \rho\ \text{kp} & = & \text{code}_V\ (e_0,\ldots,e_{k-1})\ \rho\ \text{kp} & = \\
 & & & \\
 & & & \\
 & & & \\
 & & & \\
 & & & \\
\text{code}_C\ []\ \rho\ \text{kp} & = & \text{code}_V\ []\ \rho\ \text{kp} & = \\
\text{code}_C\ (e_1 :: e_2)\ \rho\ \text{kp} & = & \text{code}_V\ (e_1 :: e_2)\ \rho\ \text{kp} & = \\
 & & & \\
 & & &
\end{array}
\begin{array}{l}
\text{code}_C\ e_0\ \rho\ \text{kp} \\
\text{code}_C\ e_1\ \rho\ (\text{kp}+1) \\
\ldots \\
\text{code}_C\ e_{k-1}\ \rho\ (\text{kp}+k-1) \\
\text{mkvec k} \\
\text{nil} \\
\text{code}_C\ e_1\ \rho\ \text{kp} \\
\text{code}_C\ e_2\ \rho\ (\text{kp}+1) \\
\text{cons}
\end{array}
$$

25 Letzte Aufrufe

Das Aufruf-Vorkommen $l \equiv e'\ e_0 \ldots e_{m-1}$ heißt letzt in einem Ausdruck e, falls die Auswertung von l den Wert für e liefern kann.

Beispiele:

$r\ t\ (h :: y)$ ist letzt in	$\textbf{match}\ x\ \textbf{with}\ [] \rightarrow y \mid h :: t \rightarrow r\ t\ (h :: y)$
$f\ (x-1)$ ist nicht letzt in	$\textbf{if}\ x \leq 1\ \textbf{then}\ 1\ \textbf{else}\ x * f\ (x-1)$

Beobachtung:

Letzte Aufrufe eines Funktions-Rumpfs benötigen keinen neuen Kellerrahmen!

$$\Longrightarrow$$

Automatische Transformation von Tail Recursion in Schleifen !!!

Der Code für einen letzten Aufruf $l \equiv (e'\ e_0 \ldots e_{m-1})$ in einer Funktion f mit k Argumenten muss:

- die aktuellen Parameter e_i anlegen und die Funktion e' bestimmen;
- die lokalen Variablen sowie die k verbrauchten Argumente von f frei geben;
- apply ausführen.

$$
\begin{array}{lll}
\text{code}_V\ l\ \rho\ \text{kp} & = & \text{code}_C\ e_{m-1}\ \rho\ \text{kp} \\
 & & \text{code}_C\ e_{m-2}\ \rho\ (\text{kp}+1) \\
 & & \ldots \\
 & & \text{code}_C\ e_0\ \rho\ (\text{kp}+m-1) \\
 & & \text{code}_V\ e'\ \rho\ (\text{kp}+m) \qquad // \text{ Auswerten der Funktion} \\
 & & \text{move}\ r\ (m+1) \qquad // \text{ Freigabe} \\
 & & \text{apply}
\end{array}
$$

wobei $r = kp + k$ die Anzahl der freizugebenden stack-Zellen ist.

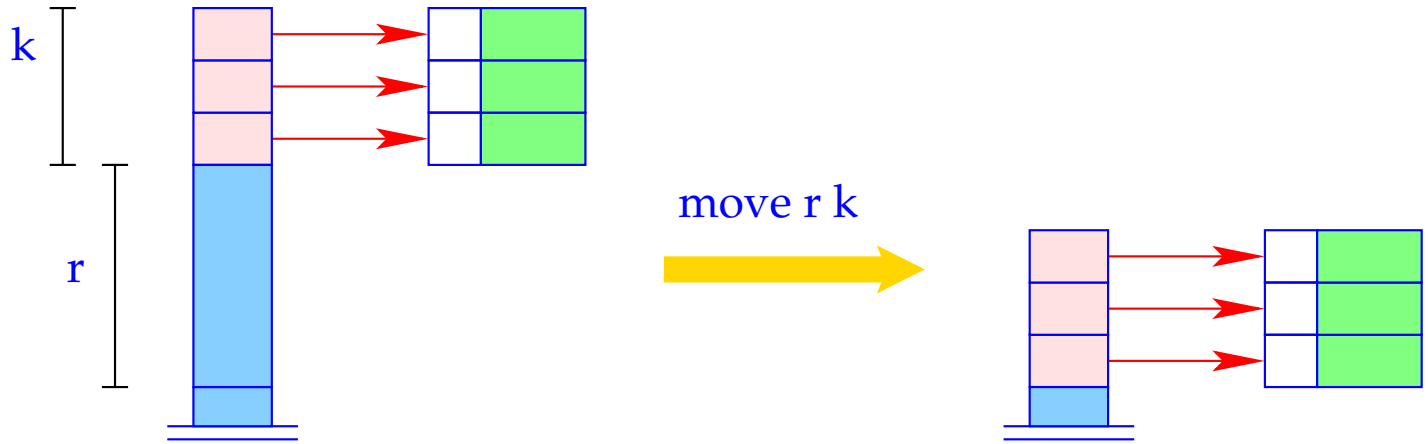
Beispiel:

Der Rumpf der Funktion

$$r = \textbf{fun}\, x, y \;\rightarrow\; \textbf{match}\, x\, \textbf{with}\, [] \;\rightarrow y \;\mid\; h :: t \;\rightarrow r\; t\; (h :: y)$$

```
0   targ 2          1          jump B          4          pushglob 0
0   pushloc 0                                  5          eval
1   eval            2   A:     pushloc 1       5          move 4 3
1   tlist A         3          pushloc 4                  apply
0   pushloc 1       4          cons                       slide 2
1   eval            3          pushloc 1       1   B:     return 2
```

Da der alte Kellerrahmen beibehalten wird, wird return 2 nur über den direkten Sprung am Ende der []-Alternative erreicht.




```
SP = SP – k – r;
for (i=1; i≤k; i++)
    S[SP+i] = S[SP+i+r];
SP = SP + k;
```

Die Übersetzung logischer Programmiersprachen

26 Die Sprache PuP

Wir betrachten hier nur die Mini-Sprache PuP (“Pure Prolog”). Insbesondere verzichten wir auf:

- Arithmetik;
- den Cut-Operator (vorerst :-)
- Selbst-Modifikation von Programmen mittels assert und retract.

Beispiel:

$$
\begin{array}{lcl}
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{elephant}, Y = \mathit{horse} \\
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{horse}, Y = \mathit{donkey} \\
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{donkey}, Y = \mathit{dog} \\
\mathsf{bigger}(X, Y) & \leftarrow & X = \mathit{donkey}, Y = \mathit{monkey} \\
\mathsf{is\_bigger}(X, Y) & \leftarrow & \mathsf{bigger}(X, Y) \\
\mathsf{is\_bigger}(X, Y) & \leftarrow & \mathsf{bigger}(X, Z), \mathsf{is\_bigger}(Z, Y) \\
? \quad \mathsf{is\_bigger}(\mathit{elephant}, \mathit{dog}) & &
\end{array}
$$

Ein realistischeres Beispiel:

$$
\begin{array}{lcl}
\mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\ Y = Z \\
\mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X', Y, Z') \\
? \quad \mathsf{app}(X, [Y, c], [a, b, Z]) & &
\end{array}
$$

Ein realistischeres Beispiel:

$$
\begin{array}{lll}
\mathsf{app}(X, Y, Z) & \leftarrow & X = [\,],\ Y = Z \\
\mathsf{app}(X, Y, Z) & \leftarrow & X = [H|X'],\ Z = [H|Z'],\ \mathsf{app}(X', Y, Z') \\
? \quad \mathsf{app}(X, [Y, c], [a, b, Z]) & &
\end{array}
$$

Bemerkung:

$[\,]$	$==$	das Atom leere Liste
$[H\|Z]$	$==$	binäre Konstructor-Anwendung
$[a, b, Z]$	$==$	Abkürzung für: $[a\|[b\|[Z\|[\,]]]]$

Ein Programm p ist darum wie folgt aufgebaut:

$$
\begin{array}{lcl}
t & ::= & a \mid X \mid \_ \mid f(t_1, \ldots, t_n) \\
g & ::= & p(t_1, \ldots, t_k) \mid X = t \\
c & ::= & p(X_1, \ldots, X_k) \leftarrow g_1, \ldots, g_r \\
p & ::= & c_1 \ldots . c_m ? g
\end{array}
$$

- Ein Term t ist entweder ein Atom, eine (evt. anonyme) Variable oder eine Konstruktor-Anwendung.
- Ein Ziel g ist entweder ein Literal, d.h. ein Prädikats-Aufruf, oder eine Unifikation.
- Eine Klausel c besteht aus einem Kopf $p(X_1, \ldots, X_k)$ mit Prädikats-Namen und Liste der formalen Parameter sowie einer Folge von Zielen als Rumpf.
- Ein Programm besteht aus einer Folge von Klauseln sowie einem Ziel als Anfrage.

Prozedurale Sicht auf PuP-Programme:

Ziel	$=\!=$	Prozedur-Aufruf
Prädikat	$=\!=$	Prozedur
Definition	$=\!=$	Rumpf
Term	$=\!=$	Wert
Unifikation	$=\!=$	elementarer Berechnungschritt
Bindung von Variablen	$=\!=$	Seiteneffekt

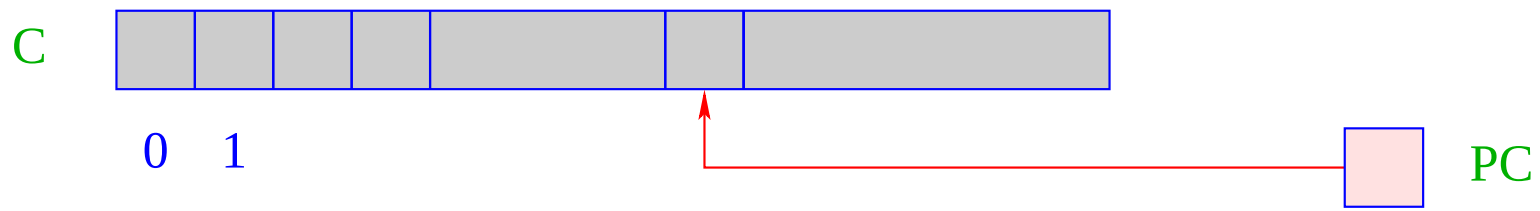
Achtung: Prädikat-Aufrufe ...

- ... liefern keinen Rückgabewert!
- ... beeinflussen den Aufrufer einzig durch Seiteneffekte :-)
- ... können fehlschlagen. Dann wird die nächste Definition probiert :-))

$\Longrightarrow$ backtracking

27 Architektur der WiM:

Der Code-Speicher:



C = Code Speicher – enthält WiM-Programm;
jede Zelle enthält einen Befehl;

PC = Program Counter – zeigt auf nächsten auszuführenden Befehl.