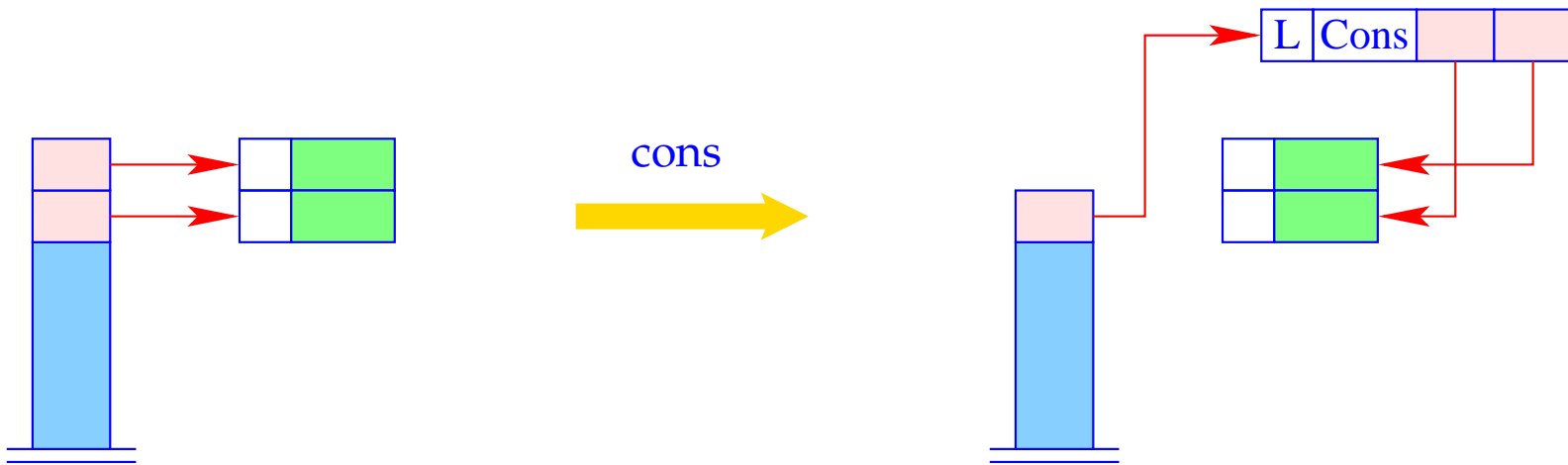


$S[SP] = SP++$ ;  $S[SP] = \text{new}(L, \text{Nil})$ ;



$S[SP-1] = \text{new } (L, \text{Cons}, S[SP-1], S[SP]);$   
 $SP--;$

## 24.4 Pattern-Matching

Betrachte den Ausdruck  $e \equiv \mathbf{case} \ e_0 \ \mathbf{of} \ [] \rightarrow e_1; \ h : t \rightarrow e_2$ .

Auswertung von  $e$  erfordert:

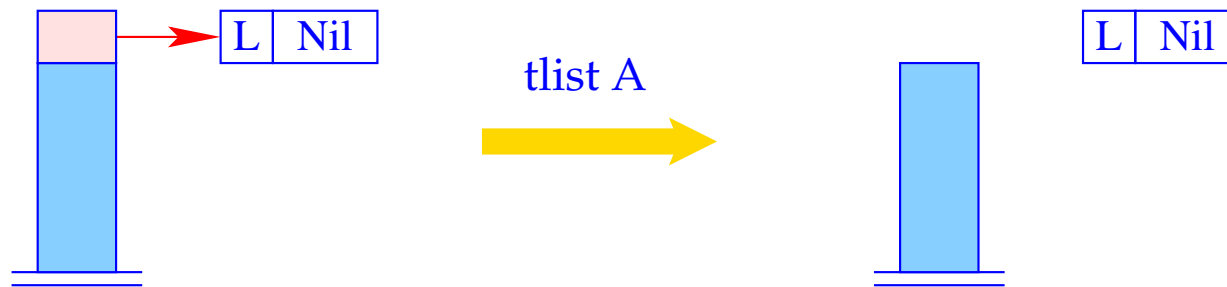
- Auswertung von  $e_0$ ;
- Überprüfung, ob  $e_0$  ein L-Objekt ist;
- Falls  $e_0$  gleich der leeren Liste ist, Auswertung von  $e_1$  ...
- ... andernfalls Kellern der Verweise des L-Objekts und Auswertung von  $e_2$ .

Folglich erhalten wir (für CBN wie CBV):

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

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

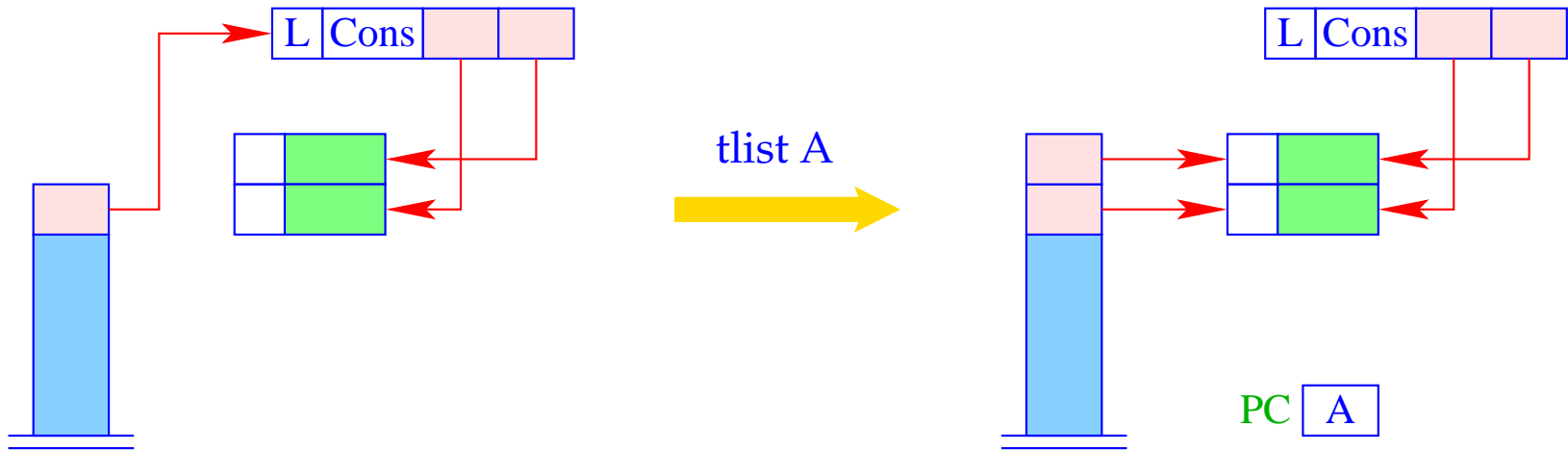
Der neue Befehl `tlist A` führt die notwendigen Überprüfungen durch und legt (im Cons-Fall) zwei neue lokale Variablen an:



```

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	mkcloc 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  $\text{code}_C$  lässt sich auch bei Tupeln und Listen optimieren:

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



## 25 Letzte Aufrufe

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

Beispiele:

$rt(h : y)$  ist **letzt** in `case x of [] → y; h : t → rt(h : y)`  
 $f(x - 1)$  ist **nicht letzt** in `if x ≤ 1 then 1 else x * f(x - 1)`

Beobachtung:

Letzte Aufrufe eines Funktions-Rumpfs benötigen **keinen neuen** Kellerrahmen!



Automatische Transformation von Tail Recursion in Schleifen !!!

Der Code für einen letzten Aufruf  $l \equiv (e' e_0 \dots e_{m-1})$  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.

```

codeV l ρ kp = codeC em-1 ρ kp
                codeC em-2 ρ (kp + 1)
                ...
                codeC e0 ρ (kp + m - 1)
                codeV e' ρ (kp + m)           // Auswerten der Funktion
                move r (m + 1)                // Freigabe
                apply

```

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

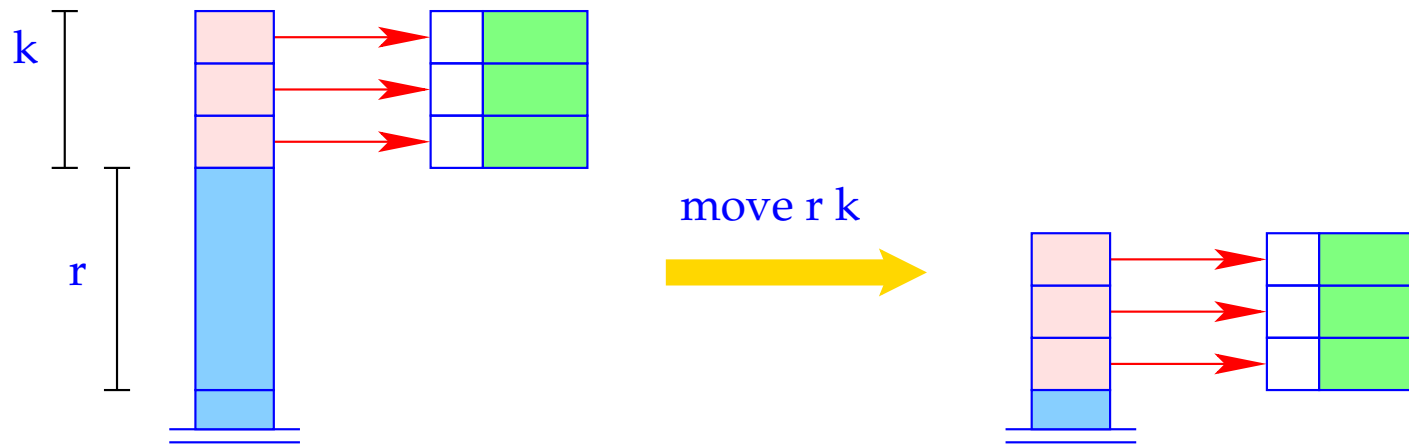
## Beispiel:

Der Rumpf der Funktion

$$r = \mathbf{fn} \ x, y \Rightarrow \mathbf{case} \ x \ \mathbf{of} \ [] \rightarrow y; \ h : t \rightarrow r \ t \ (h : y)$$

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:

$\text{bigger}(X, Y) \leftarrow X = \textit{elephant}, Y = \textit{horse}$

$\text{bigger}(X, Y) \leftarrow X = \textit{horse}, Y = \textit{donkey}$

$\text{bigger}(X, Y) \leftarrow X = \textit{donkey}, Y = \textit{dog}$

$\text{bigger}(X, Y) \leftarrow X = \textit{donkey}, Y = \textit{monkey}$

$\text{is\_bigger}(X, Y) \leftarrow \text{bigger}(X, Y)$

$\text{is\_bigger}(X, Y) \leftarrow \text{bigger}(X, Z), \text{is\_bigger}(Z, Y)$

?  $\text{is\_bigger}(\textit{elephant}, \textit{dog})$

## Ein realistischeres Beispiel:

$$\text{app}(X, Y, Z) \leftarrow X = [], Y = Z$$

$$\text{app}(X, Y, Z) \leftarrow X = [H|X'], Z = [H|Z'], \text{app}(X', Y, Z')$$

$$? \text{app}(X, [Y, c], [a, b, Z])$$



## Ein realistischeres Beispiel:

$\text{app}(X, Y, Z) \leftarrow X = [], Y = Z$

$\text{app}(X, Y, Z) \leftarrow X = [H|X'], Z = [H|Z'], \text{app}(X', Y, Z')$

?  $\text{app}(X, [Y, c], [a, b, Z])$

## Bemerkung:

$[]$   $\equiv$  das Atom **leere Liste**

$[H|Z]$   $\equiv$  **binäre** Constructor-Anwendung

$[a, b, Z]$   $\equiv$  Abkürzung für:  $[a|[b|[Z|[ ]]]]$