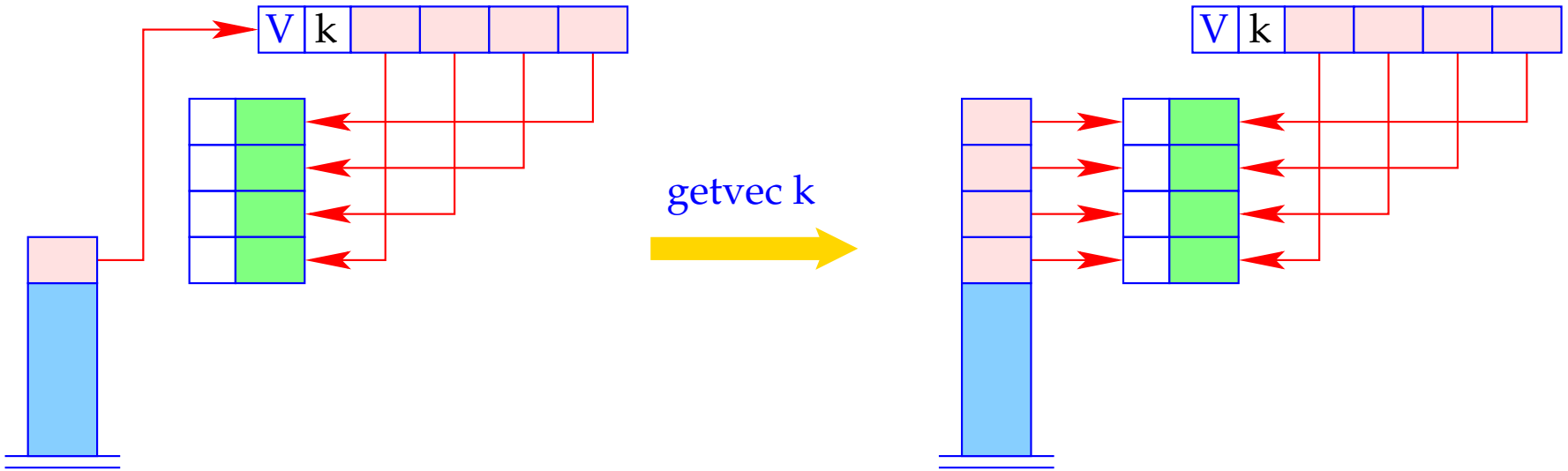
Möchte man nicht auf einzelne, sondern gegebenenfalls alle Komponenten eines Tupels zugreifen, kann man dies mithilfe des Ausdrucks
$e \equiv \textbf{let}\ (y_0, \ldots, y_{k-1}) = e_1\ \textbf{in}\ e_0$ tun.

Diesen übersetzen wir wie folgt:

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

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

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

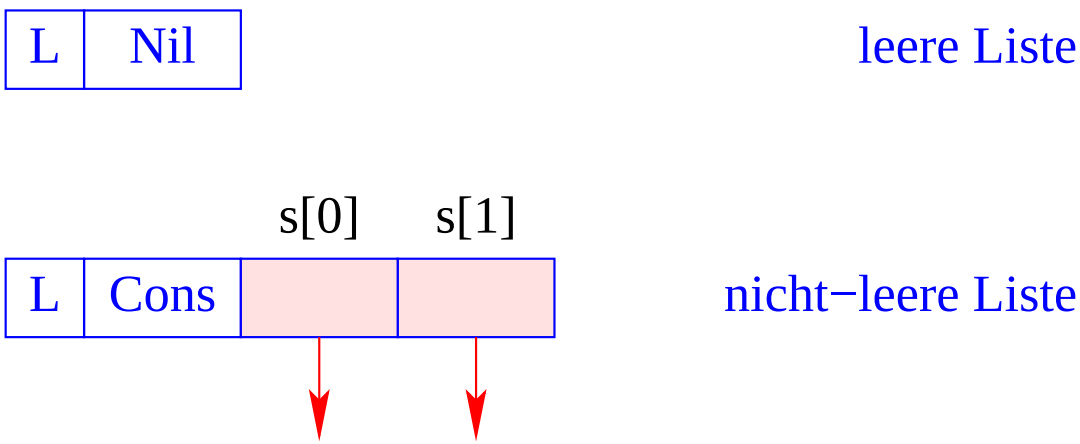
Beispiel: Die Append-Funktion app:

$$\begin{array}{lcl} \text{app} & = & \textbf{fn}\ l, y \Rightarrow \textbf{case}\ l\ \textbf{of} \\ & & \quad [] \quad \rightarrow \quad y \\ & & \quad h : t \quad \rightarrow \quad h : (\text{app}\ t\ y) \end{array}$$

Folglich erweitern wir die Syntax von Ausdrücken e um:

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

Neue Halden-Objekte:



24.3 Der Aufbau von Listen

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

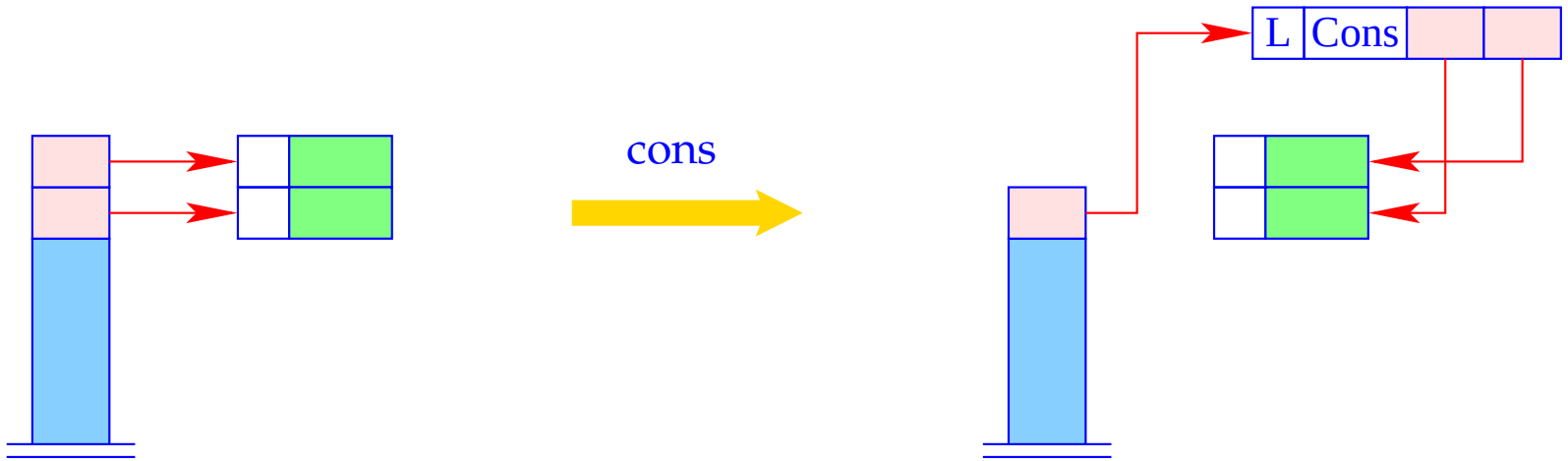
Damit erhalten wir für CBN:

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

Beachte:

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

S[SP] = 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 $e \equiv \textbf{case}\ e_0\ \textbf{of}\ [] \rightarrow e_1;\ \ h : t \rightarrow e_2$.

Auswertung von e erfordert:

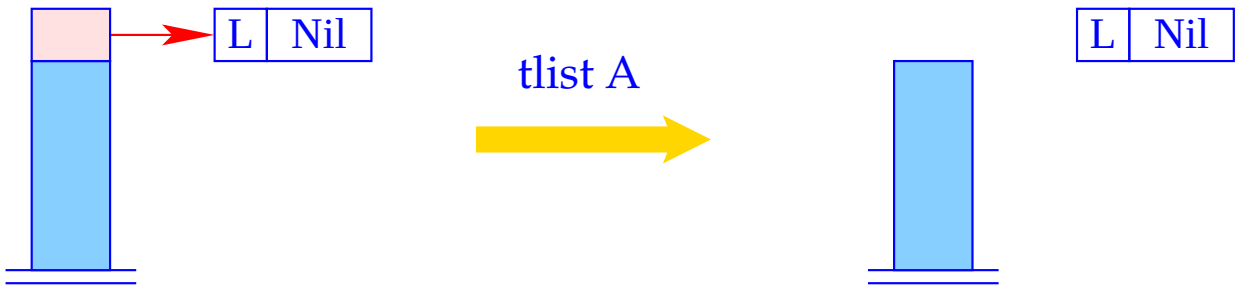
- Auswertung von e_0;
- Überprüfung, ob e_0 ein L-Objekt ist;
- Falls e_0 gleich der leeren Liste ist, Auswertung von e_1 ...
- ... andernfalls Kellern der Verweise des L-Objekts und Auswertung von e_2.

Folglich erhalten wir (für CBN wie CBV):

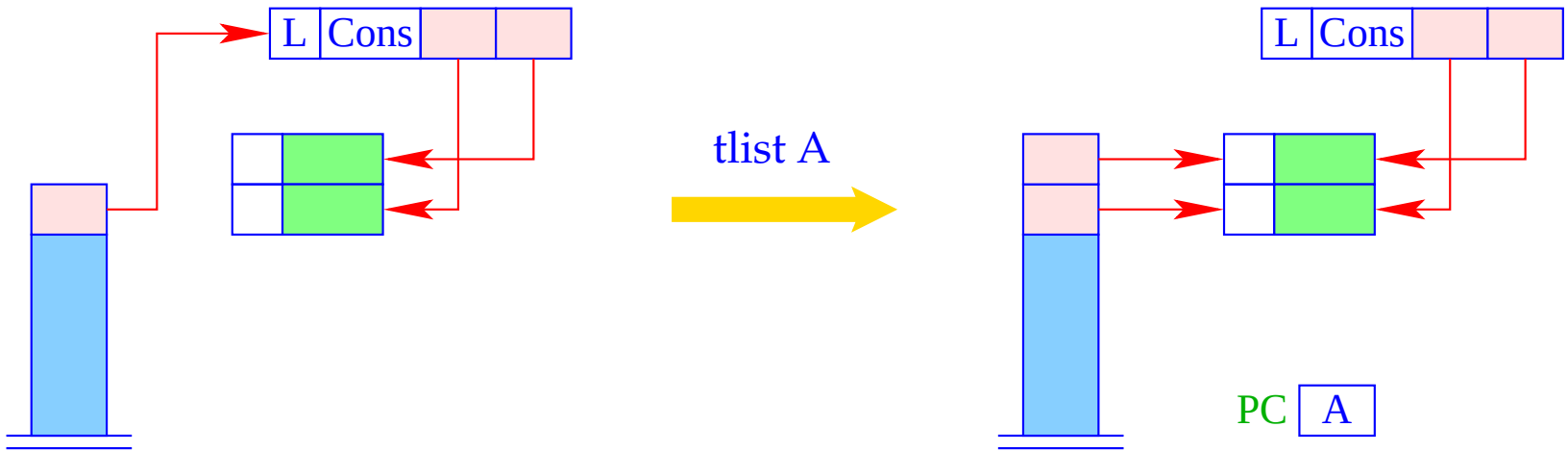
$$
\begin{array}{lll}
\text{code}_V\ e\ \rho\ \text{kp} & = & \text{code}_V\ e_0\ \rho\ \text{kp} \\
 & & \text{tlist A} \\
 & & \text{code}_V\ e_1\ \rho\ \text{kp} \\
 & & \text{jump B} \\
 & \text{A :} & \text{code}_V\ e_2\ \rho'\ (\text{kp} + 2) \\
 & & \text{slide 2} \\
 & \text{B :} & \ldots
\end{array}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

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:

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

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	$\textbf{case}\ x\ \textbf{of}\ [] \rightarrow y;\ \ h : t \rightarrow r\ t\ (h : y)$
$f\ (x - 1)$ 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.

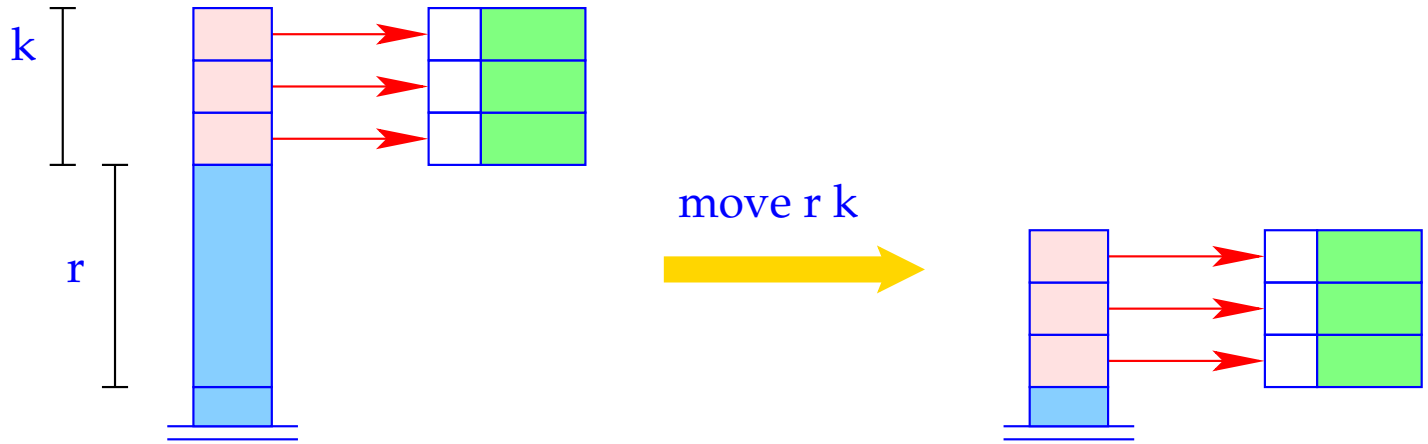
Beispiel:

Der Rumpf der Funktion

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

```
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.