## 24.5 Closures of Tuples and Lists

The general schema for $\text{code}_C$ can be optimized for tuples and lists:

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

# 25 Last Calls

A function application is called last call in an expression $e$ if this application could deliver the value for $e$.

A last call usually is the outermost application of a defining expression.

A function definition is called tail recursive if all recursive calls are last calls.

## Examples:

$r\ t\ (h : y)$ is a last call in $\quad$ $\textbf{case}\ x\ \textbf{of}\ [\,]\ \to y;\ \ h : t\ \to r\ t\ (h : y)$

$f\ (x - 1)$ is not a last call in $\quad$ $\textbf{if}\ x \leq 1\ \textbf{then}\ 1\ \textbf{else}\ x * f\ (x - 1)$

**Observation:** Last calls in a function body need no new stack frame!

$$\Longrightarrow$$

Automatic transformation of tail recursion into loops!!!

The code for a last call $l \equiv (e' \; e_0 \ldots e_{m_1})$ inside a function $f$ with $k$ arguments must

1. allocate the arguments $e_i$ and evaluate $e'$ to a function (note: all this inside $f$'s frame!);
2. deallocate the local variables and the $k$ consumed arguments of $f$;
3. execute an apply.

$$
\begin{array}{lcll}
\mathrm{code}_V \; l \; \rho \; \mathrm{sd} & = & \mathrm{code}_C \; e_{m-1} \; \rho \; \mathrm{sd} & \\
 & & \mathrm{code}_C \; e_{m-2} \; \rho \; (\mathrm{sd}+1) & \\
 & & \ldots & \\
 & & \mathrm{code}_C \; e_0 \; \rho \; (\mathrm{sd}+m-1) & \\
 & & \mathrm{code}_V \; e' \; \rho \; (\mathrm{sd}+m) & // \text{ Evaluation of the function} \\
 & & \mathrm{move} \; r \; (m+1) & // \text{ Deallocation of } r \text{ cells} \\
 & & \mathrm{apply} &
\end{array}
$$

where $r = sd + k$ is the number of stack cells to deallocate.

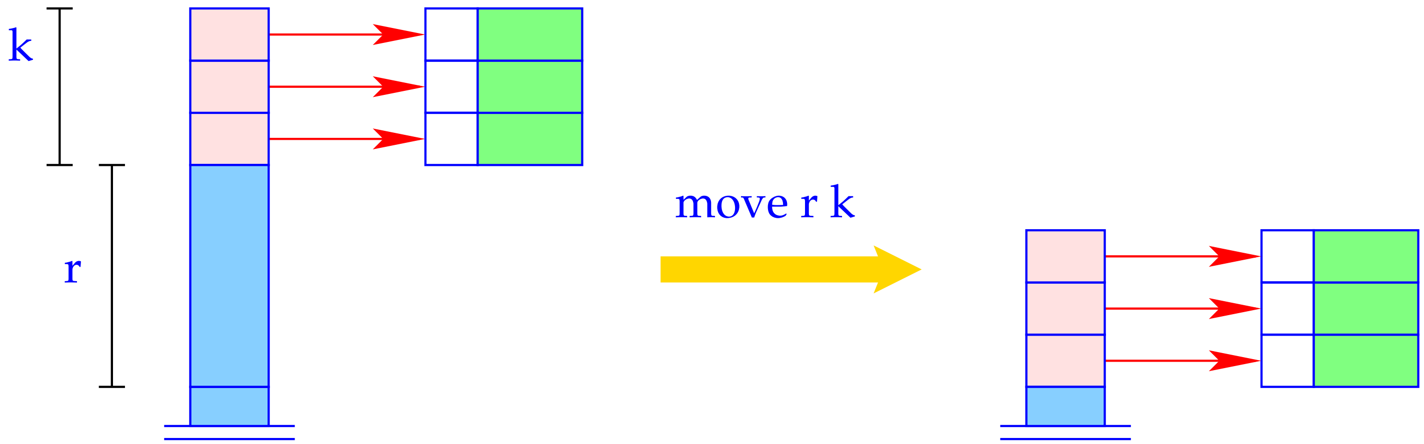
## Example:

The body of the function

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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 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 |

Since the old stack frame is kept, return 2 will only be reached by the direct jump at the end of the []-alternative.




$$
\begin{array}{l}
SP = SP - k - r; \\
\text{for } (i{=}1;\ i \leq k;\ i{+}{+}) \\
\quad S[SP+i] = S[SP+i+r]; \\
SP = SP + k;
\end{array}
$$

# The Translation of Logic Languages

# 26 The Language Proll

Here, we just consider the core language Proll (“Prolog-light” :-). In particular, we omit:

- arithmetic;
- the cut operator;
- self-modification of programs through assert and retract.

Example:

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

## A More Realistic Example:

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

## A More Realistic Example:

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

## Remark:

| | | |
|---|---|---|
| $[\,]$ | $==$ | the atom empty list |
| $[H\|Z]$ | $==$ | binary constructor application |
| $[a,b,Z]$ | $==$ | shortcut for: $[a\|[b\|[Z\|[\,]]]]$ |

A program $p$ is constructed as follows:

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

- A term $t$ either is an atom, a variable, an anonymous variable or a constructor application.
- A goal $g$ either is a literal, i.e., a predicate call, or a unification.
- A clause $c$ consists of a head $p(X_1, \ldots, X_k)$ with predicate name and list of formal parameters together with a body, i.e., a sequence of goals.
- A program consists of a sequence of clauses together with a single goal as query.

## Procedural View of Proll programs:

| | | |
|---|---|---|
| goal | $==$ | procedure call |
| predicate | $==$ | procedure |
| clause | $==$ | definition |
| term | $==$ | value |
| unification | $==$ | basic computation step |
| binding of variables | $==$ | side effect |

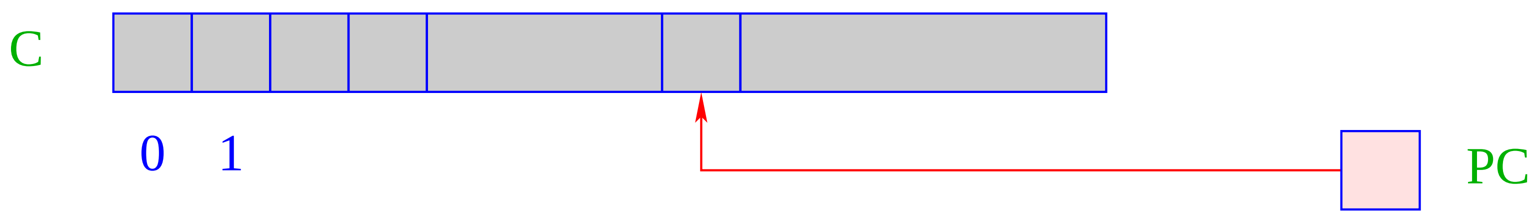
## Note: Predicate calls ...

- ... do not have a return value.
- ... affect the caller through side effects only :-)
- ... may fail. Then the next definition is tried :-))

$\Longrightarrow$ backtracking

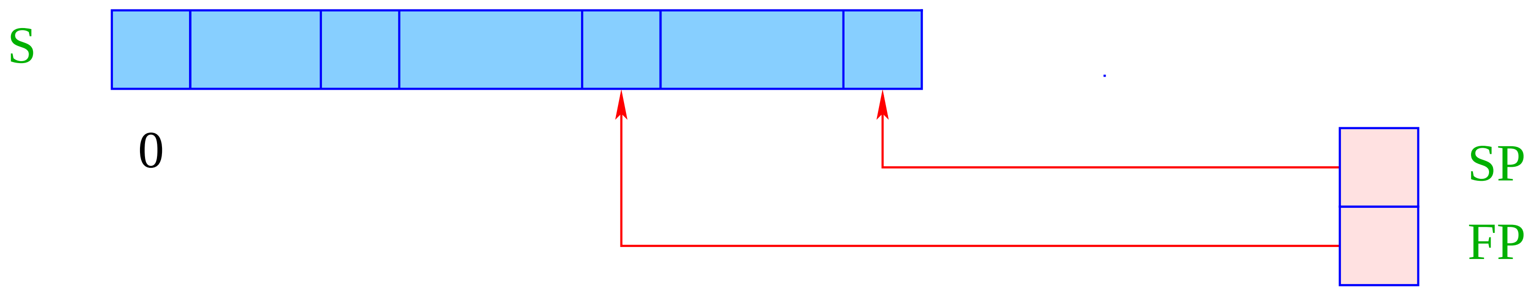
# 27 Architecture of the WiM:

## The Code Store:



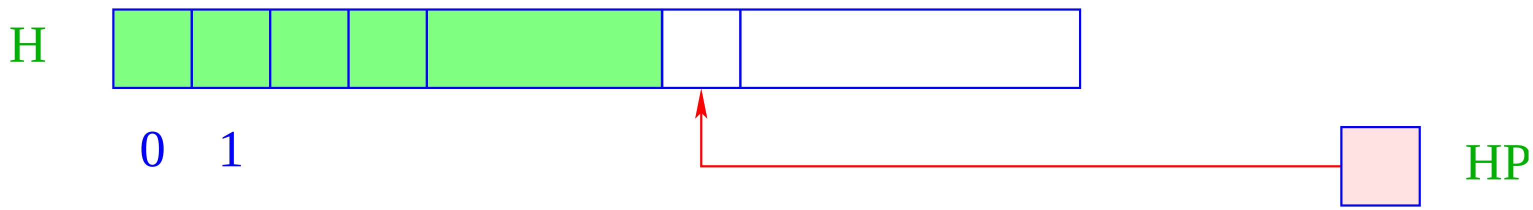

| | | |
|---|---|---|
| C | = | Code store – contains WiM program; |
| | | every cell contains one instruction; |
| PC | = | Program Counter – points to the next instruction to executed; |

## The Runtime Stack:



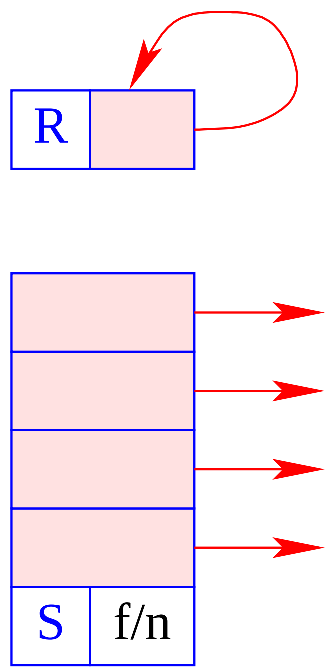

| | | |
|---|---|---|
| S | = | Runtime Stack – every cell may contain a value or an address; |
| SP | = | Stack Pointer – points to the topmost occupied cell; |
| FP | = | Frame Pointer – points to the current stack frame. |
| | | Frames are created for predicate calls, |
| | | contain cells for each variable of the current clause |

# The Heap:




| | | |
|---|---|---|
| H | = | Heap for dynamicly constructed terms; |
| HP | = | Heap-Pointer – points to the first free cell; |

- The heap in maintained like a stack as well :-)
- A new-instruction allocates a object in H.
- Objects are tagged with their types (as in the MaMa) ...




| | |
|---|---|
| atom | 1 cell |
| variable | 1 cell |
| unbound variable | 1 cell |
| structure | (n+1) cells |

# 28 Construction of Terms in the Heap

Parameter terms of goals (calls) are constructed in the heap before passing.

Assume that the address environment $\rho$ returns, for each clause variable $X$ its address (relative to FP) on the stack. Then $\quad \text{code}_A\ t\ \rho \quad$ should ...

- construct (a presentation of) $t$ in the heap; and
- return a reference to it on top of the stack.

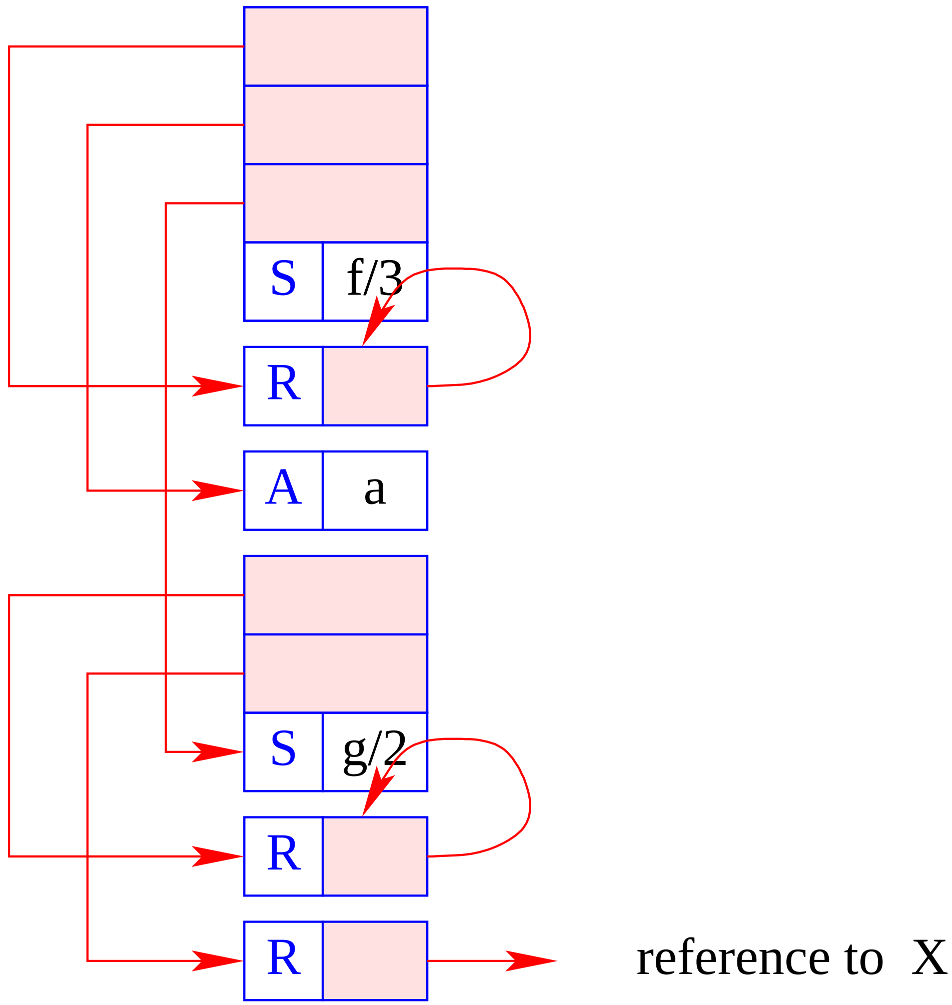
## Idea:

- Construct the tree during a post-order traversal of $t$
- with one instruction for each new node!

**Example:** $\qquad t \equiv f(g(X, Y), a, Z).$

Assume that $X$ is initialized, i.e., $\quad S[FP + \rho\, X] \quad$ contains already a reference, $Y$ and $Z$ are not yet initialized.

Representing $t \equiv f(g(X,Y),a,Z)$ :



For a distinction, we mark occurrences of already initialized variables through over-lining (e.g. $\bar{X}$).

Note: Arguments are always initialized!

Then we define:

$$\begin{array}{lll}
\text{code}_A\ a\ \rho & = & \text{putatom a} \\
\text{code}_A\ X\ \rho & = & \text{putvar}\ (\rho\, X) \\
\text{code}_A\ \bar{X}\ \rho & = & \text{putref}\ (\rho\, X) \\
\text{code}_A\ _\ \rho & = & \text{putanon}
\end{array}
\qquad
\begin{array}{lll}
\text{code}_A\ f(t_1, \ldots, t_n)\ \rho & = & \text{code}_A\ t_1\ \rho \\
 & & \ldots \\
 & & \text{code}_A\ t_n\ \rho \\
 & & \text{putstruct f/n}
\end{array}$$

For a distinction, we mark occurrences of already initialized variables through over-lining (e.g. $\bar{X}$).

Note: Arguments are always initialized!

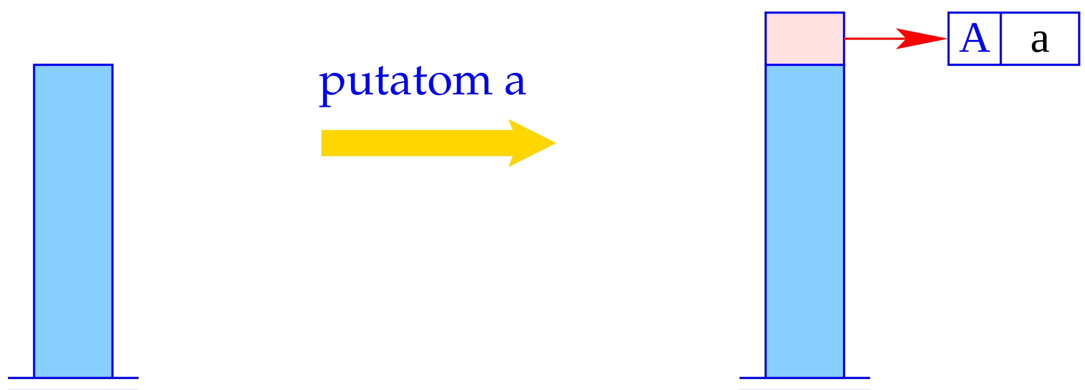
Then we define:

$$
\begin{array}{lll}
\text{code}_A\, a\, \rho & = & \text{putatom a} \\
\text{code}_A\, X\, \rho & = & \text{putvar}\,(\rho\, X) \\
\text{code}_A\, \bar{X}\, \rho & = & \text{putref}\,(\rho\, X) \\
\text{code}_A\, \_\, \rho & = & \text{putanon}
\end{array}
\qquad
\begin{array}{lll}
\text{code}_A\, f(t_1, \ldots, t_n)\, \rho & = & \text{code}_A\, t_1\, \rho \\
 & & \ldots \\
 & & \text{code}_A\, t_n\, \rho \\
 & & \text{putstruct f/n}
\end{array}
$$

For $f(g(\overline{X}, Y), a, Z)$ and $\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$ this results in the sequence:

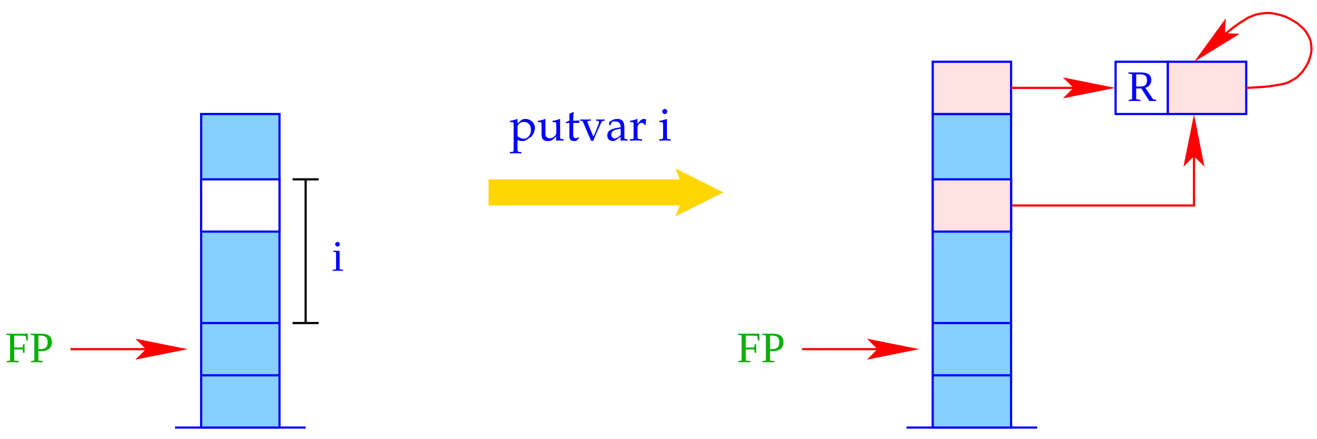
| | |
|---|---|
| putref 1 | putatom a |
| putvar 2 | putvar 3 |
| putstruct g/2 | putstruct f/3 |

The instruction putatom a constructs an atom in the heap:



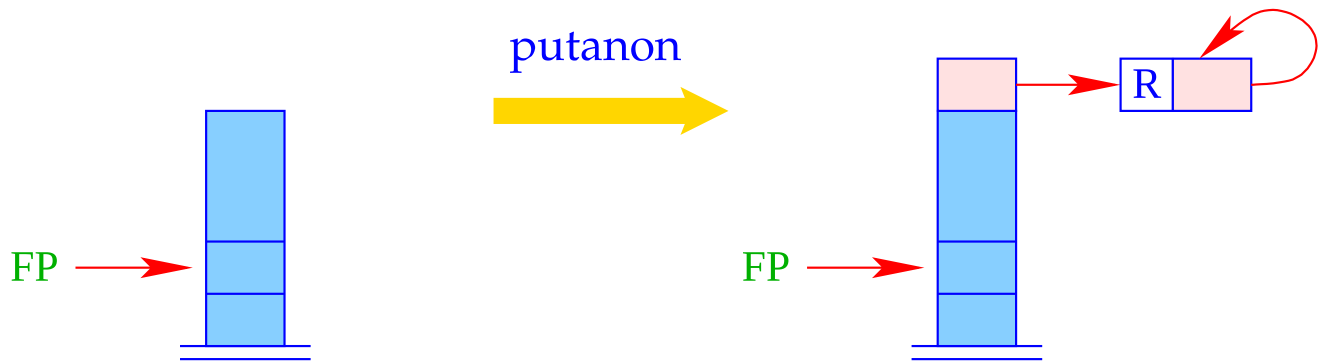

SP++; S[SP] = new (A,a);

The instruction putvar i introduces a new unbound variable and additionally initializes the corresponding cell in the stack frame:



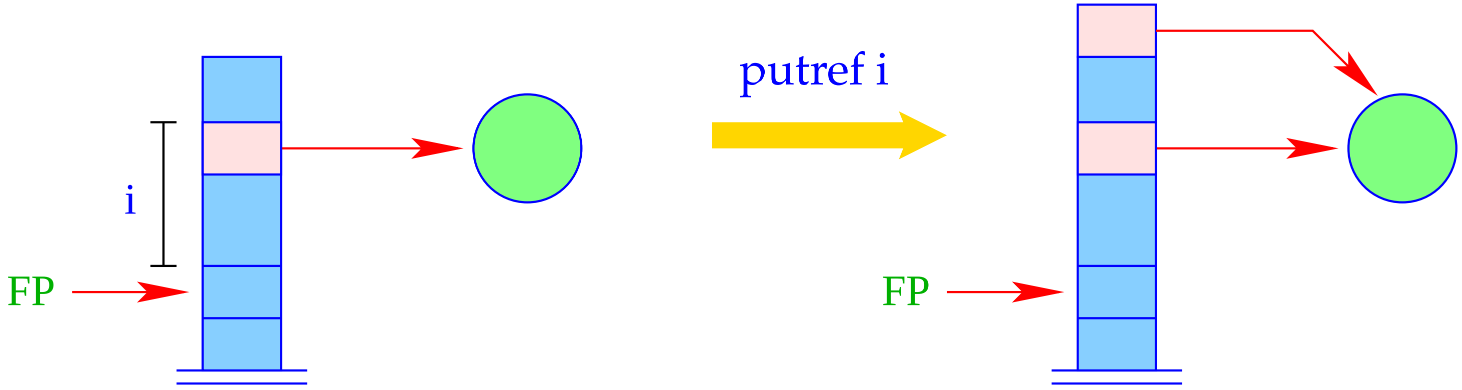

SP = SP + 1;
S[SP] = new (R, HP);
S[FP + i] = S[SP];

The instruction putanon introduces a new unbound variable but does not store a reference to it in the stack frame:



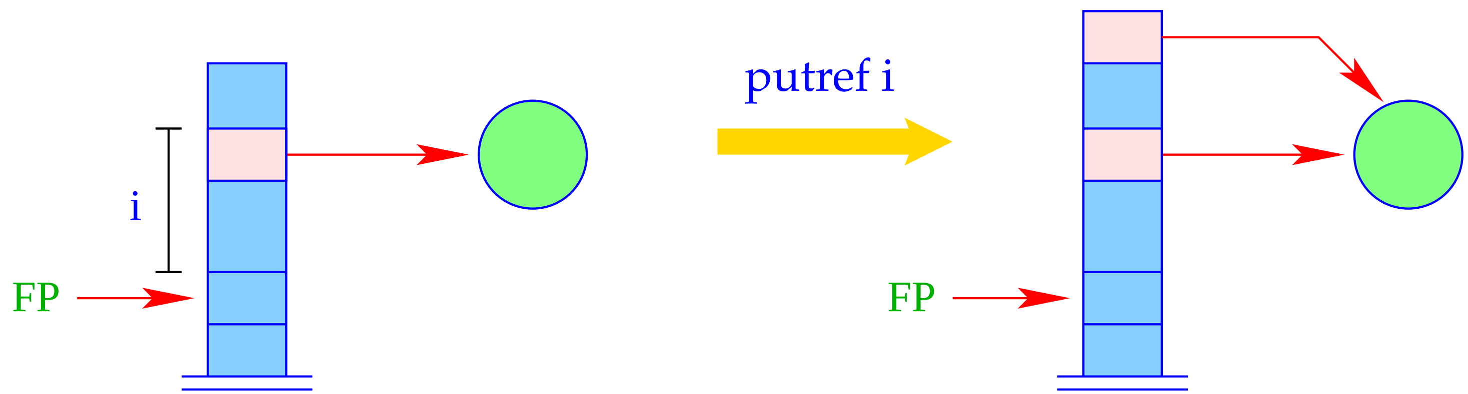

SP = SP + 1;
S[SP] = new (R, HP);

The instruction putref i pushes the value of the variable onto the stack:




SP = SP + 1;
S[SP] = deref S[FP + i];

The instruction putref i pushes the value of the variable onto the stack:

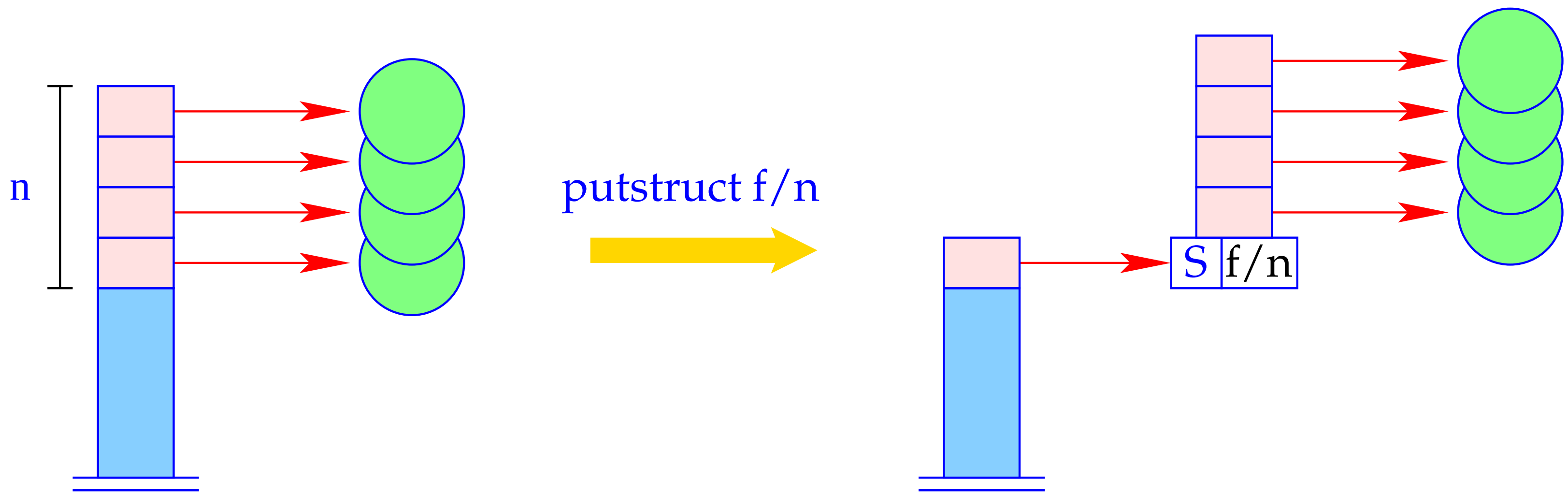



SP = SP + 1;
S[SP] = deref S[FP + i];

The auxiliary function deref contracts chains of references:

```
ref deref (ref v) {
      if (H[v]==(R,w) && v!=w) return deref (w);
      else return v;
}
```

The instruction putstruct f/n builds a constructor application in the heap:




```
v = new (S, f, n);
SP = SP - n + 1;
for (i=1; i<=n; i++)
    H[v + i] = S[SP + i -1];
S[SP] = v;
```

## Remarks:

- The instruction putref i does not just push the reference from $S[FP + i]$ onto the stack, but also dereferences it as much as possible

  $\Longrightarrow$ maximal contraction of reference chains.

- In constructed terms, references always point to smaller heap addresses.

  Also otherwise, this will be often the case. Sadly enough, it cannot be guaranteed in general :-(