

WiM — a simple abstract machine
for logical languages

The language Proll

We consider a small logic programming language Proll ("Prolog-light").

Compared with Prolog, we do not treat:

- arithmetic operations;
- the cut operator;
- self-modification of programs using `assert` and `retract`.

The language Prolog

A program p has the following syntax:

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

- A *term* t is either an atom (ie. constant), a variable, an anonymous variable, or a constructor application.
- A *goal* g is either literal, ie. a predicate call, or a unification.
- A *clause* c has a *head* $p(X_1, \dots, X_k)$ and a *body* (ie. a sequence of goals).
- A *program* consists of sequence of clauses together with a *query* (ie. a single top-level goal).

The language Proll

Example:

```
bigger(X, Y)      ←  X = elephant, Y = horse
bigger(X, Y)      ←  X = horse, Y = donkey
bigger(X, Y)      ←  X = donkey, Y = dog
bigger(X, Y)      ←  X = donkey, Y = monkey
is_bigger(X, Y)   ←  bigger(X, Y)
is_bigger(X, Y)   ←  bigger(X, Z), is_bigger(Z, Y)
?is_bigger(elephant, dog)
```

The language Proll

Example:

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

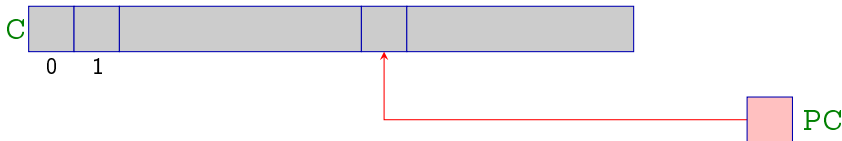
$\text{app}(X, Y, Z) \leftarrow X = [H \mid X'], Z = [H \mid Z'], \text{app}(X', Y, Z')$

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

- $[]$ the atom denoting an empty list;
- $[H \mid Z]$ a binary list constructor application;
- $[a, b, Z]$ is a shorthand of $[a \mid [b \mid [Z \mid []]]]$.

WiM architecture

Code:



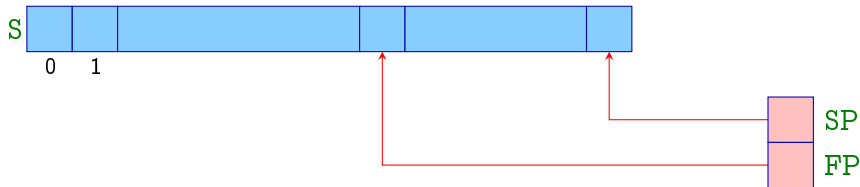
C = Code-store — memory area for a program code; each cell contains a single AM instruction.

PC = Program Counter — register containing an address of the instruction to be executed *next*.

Initially, **PC** contains the address 0; ie. **C[0]** contains the first instruction of the program.

WiM architecture

Stack:



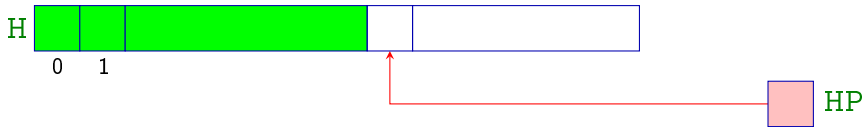
S = **S**tick — each cell contains a primitive value or an address;

SP = **S**tack-**P**ointer — points to top of the stack;

FP = **F**rame-**P**ointer — points to the currently active frame.

WiM architecture

Heap:



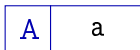
H = **H**heap — memory area for dynamically allocated data;

HP = **H**heap-**P**ointer — points to the first free cell.

- The instruction **new** creates a new object in the heap.
- Objects are tagged with their types (like in **MaMa**).

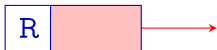
WiM architecture

Heap may contain following objects:



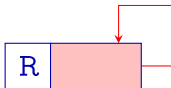
Atom

1 cell



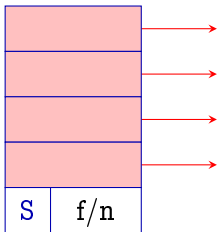
Variable

1 cell



Unbound Variable

1 cell



Structure

n+1 cells

Construction of Terms

Before parameters are passed to goals, the corresponding terms are constructed in the heap.

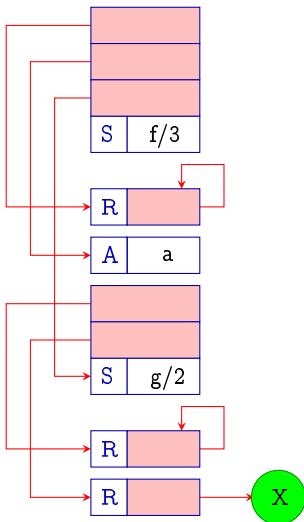
The address environment ρ binds each clause variable X with its address in the stack (relative of **FP**).

Construction of terms is performed by function $\text{code}_A t \rho$, which:

- creates a tree representation of the term t in the heap ;
- returns a pointer to it on top of the stack.

Construction of Terms

Example: Representation of the term $t \equiv f(g(X, Y), a, Z)$, where X is an initialized variable, and Y and Z are not yet initialized.



Construction of Terms

$\text{code}_A a \rho$	$=$	$\text{putatom } a$	$\text{code}_A f(t_1, \dots, t_n) \rho$	$=$
$\text{code}_A X \rho$	$=$	$\text{putvar } (\rho X)$	$\text{code}_A t_1 \rho$	
$\text{code}_A \bar{X} \rho$	$=$	$\text{putref } (\rho X)$	\dots	
$\text{code}_A - \rho$	$=$	putanon	$\text{code}_A t_n \rho$	
			$\text{putstruct } f/n$	

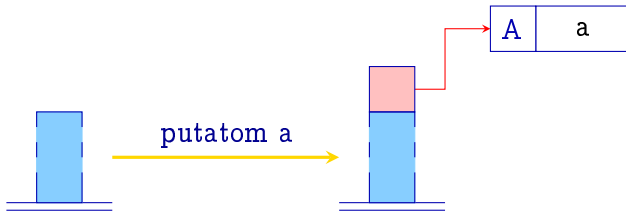
where X is an uninitialized and \bar{X} is an initialized variable.

Example: let $t \equiv f(g(\bar{X}, Y), a, Z)$ and $\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$, then

$\text{code}_A t \rho$ emits the code:

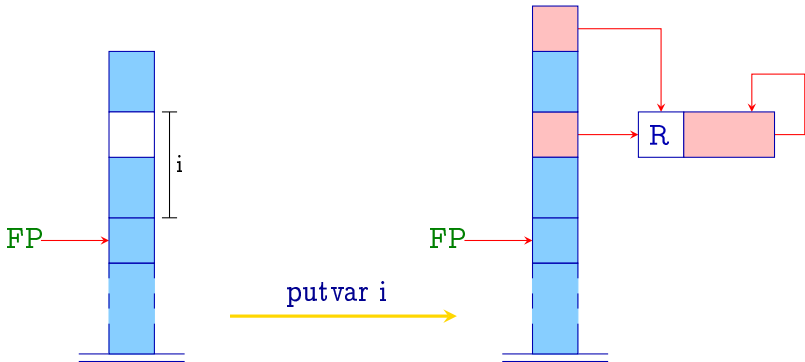
$\text{putref } 1$	$\text{putatom } a$
$\text{putvar } 2$	$\text{putvar } 3$
$\text{putstruct } g/2$	$\text{putstruct } f/3$

Construction of Terms



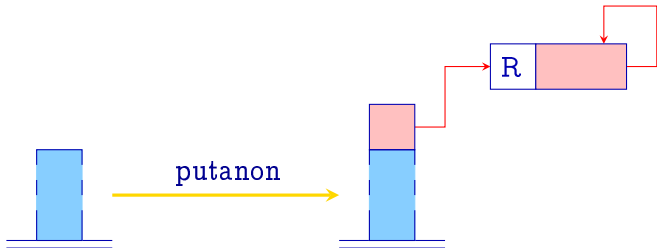
```
SP++;  
S[SP] = new (A, a);
```

Construction of Terms



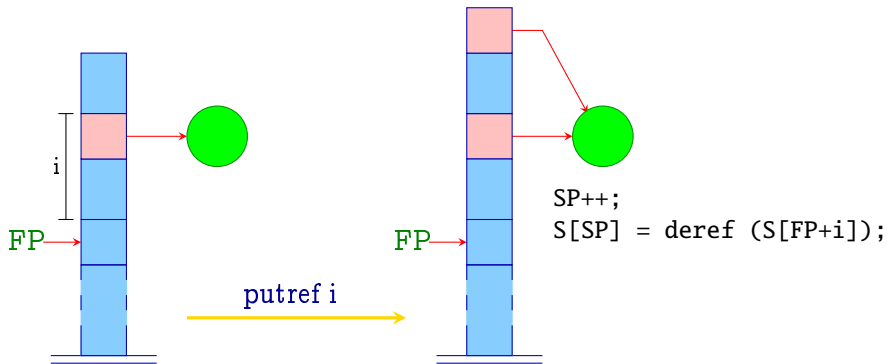
```
SP++;  
S[SP] = new (R,HP);  
S[FP+i] = S[SP];
```

Construction of Terms



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

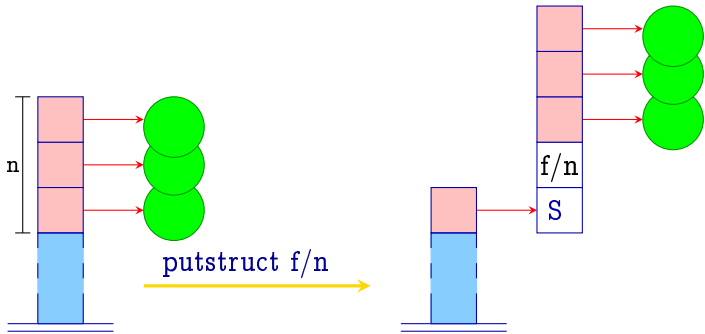
Construction of Terms



The auxiliary function `deref` contracts chains of references:

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

Construction of Terms



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

Construction of Terms

Remarks:

- The instruction `putref i` not only copies a reference from `S[FP+i]`, but also dereferences it as much as possible.
- During term construction references always point to smaller heap addresses. Even though, this is also case in many other situations, it is not guaranteed in general.

Translation of Goals

- Goals correspond to procedure calls.
- Their translation is performed by the function `codeG`.
- First create a stack frame.
- Then construct the actual parameters in the heap
- ... and store references to these into the stack frame.
- Finally, jump to the code of the predicate.

Translation of Goals

```
codeG p(t1, ..., tk) ρ =   mark A
                               codeA t1 ρ
                               ...
                               codeA tk ρ
                               call p/k
                               A: ...
```

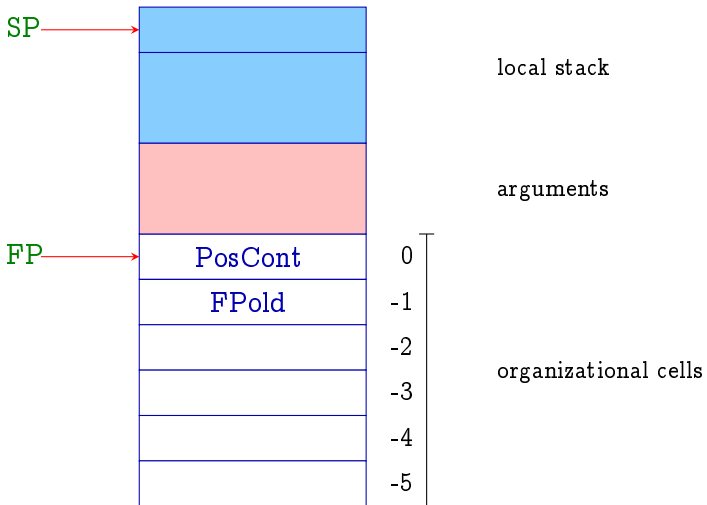
Example: let $g \equiv p(a, X, g(\bar{X}, Y))$ and $\rho = \{X \mapsto 1, Y \mapsto 2\}$,

then `codeG g ρ` emits the code:

```
mark A           putref 1           call p/3
putatom a        putvar 2           A: ...
putvar 1         putstruct g/2
```

Translation of Goals

Structure of a frame:

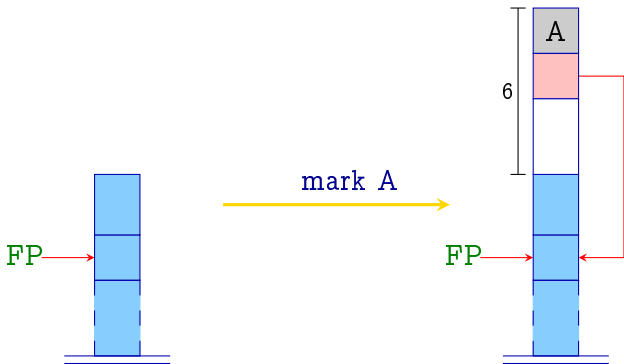


Translation of Goals

Remarks:

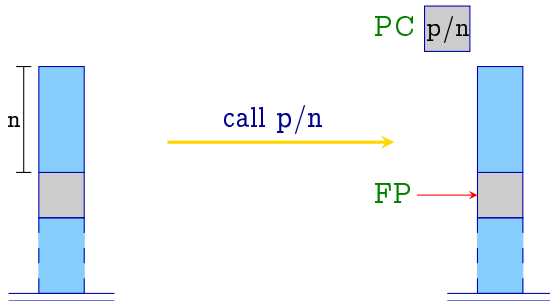
- The *positive continuation* address **PosCont** records where to continue after successful treatment of the goal.
- Additional organizational cells are necessary for *backtracking*.

Translation of Goals



$SP = SP + 6;$
 $S[SP] = A;$
 $S[SP-1] = FP;$

Translation of Goals



$$FP = SP - n;$$

$$PC = p/n;$$

Unification

- We denote occurrences of a variable X by \tilde{X} .
- It will be translated differently depending whether it's initialized or not.
- We introduce the macro `put` \tilde{X} ρ :

```
put  $X$   $\rho$  = putvar ( $\rho X$ )  
put  $\tilde{X}$   $\rho$  = putref ( $\rho X$ )  
put  $-$   $\rho$  = putanon
```

Unification

Translation of the unification $\tilde{X} = t$:

- push a reference to X onto the stack;
- construct the term t in the heap;
- introduce a new instruction which implements the unification.

$$\text{code}_G (\tilde{X} = t) \rho = \begin{array}{l} \text{put } \tilde{X} \rho \\ \text{code}_A t \rho \\ \text{unify} \end{array}$$

Unification

Example: consider the equation

$$\bar{U} = f(g(\bar{X}, Y), a, Z)$$

Then, given an address environment

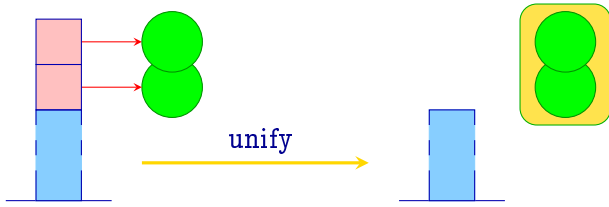
$$\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3, U \mapsto 4\}$$

the following code is generated:

```
putref 4           putatom a
putref 1           putvar 3
putvar 2           putstruct f/3
putstruct g/2      unify
```

Unification

Instruction `unify` applies the run-time function `unify()` to the topmost two references:



```
unify (S[SP-1], S[SP-2]);  
SP = SP - 2;
```

Unification

Function unify()

- ... takes two heap addresses. For each call we guarantee that these are *maximally dereferenced*.
- ... checks whether the two addresses are already *identical*. In that case does nothing and the unification succeeded.
- ... binds *younger variables* (larger addresses) to *older variables* (smaller addresses).
- ... when binding a variable to a term, checks whether the variable occurs inside the term (*occur-check*).
- ... records newly created bindings.
- ... may fail, in which case initiates *backtracking*.

Unification

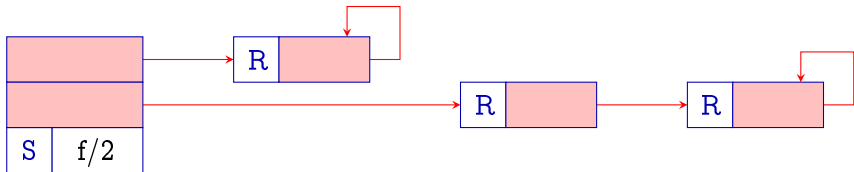
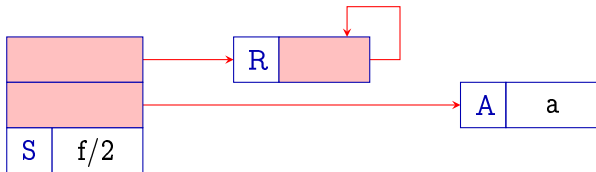
```
bool unify (ref u, ref v) {
    if (u == v) return true;
    if (H[u] == (R,_)) {
        if (H[v] == (R,_)) {
            if (u > v) {
                H[u] = (R,v); trail(u); return true;
            } else {
                H[v] = (R,u); trail(v); return true;
            }
        } else if (check (u,v)) {
            H[u] = (R,v); trail(u); return true;
        } else { backtrack(); return false; }
    }
}
...
```

Unification

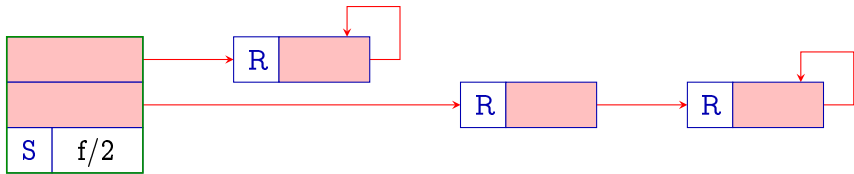
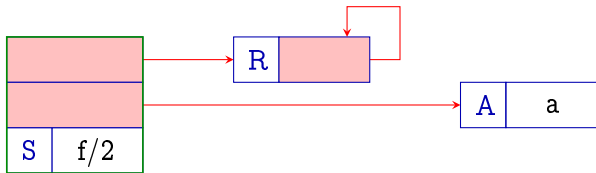
...

```
if (H[v] == (R,-)) {
    if (check (v,u)) {
        H[v] = (R,u); trail(v); return true;
    } else { backtrack(); return false; }
}
if (H[u] == (A,a) && H[v] == (A,a)) return true;
if (H[u] == (S,f/n) && H[v] == (S,f/n)) {
    for (int i=1; i<=n; i++)
        if (!unify (deref(H[u+i]), deref(H[v+i])))
            return false;
    return true;
}
backtrack(); return false;
}
```

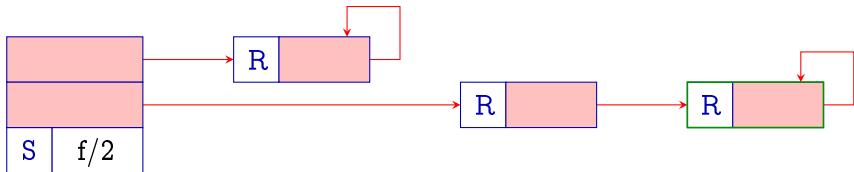
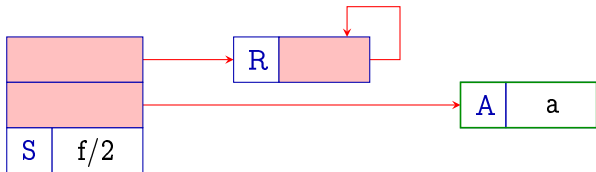
Unification



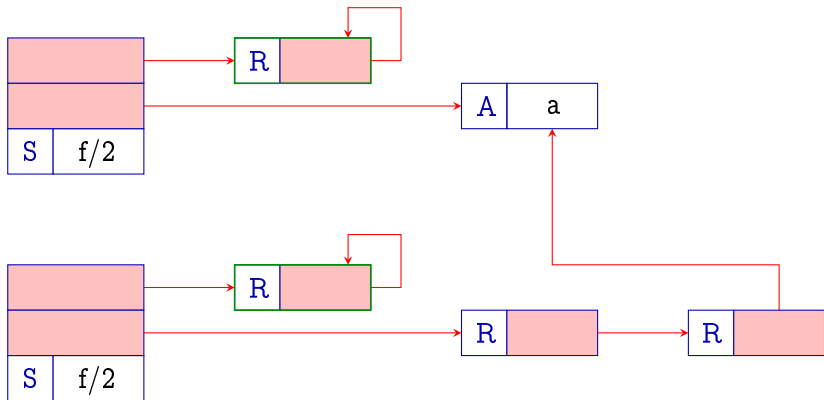
Unification



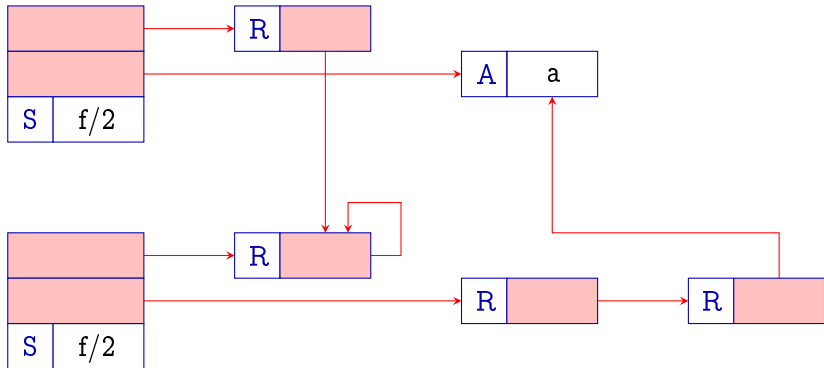
Unification



Unification



Unification



Unification

- The function `trail()` records new bindings.
- The function `backtrack()` initiates backtracking.
- The function `check()` performs the occur-check; ie. tests whether a variable (its first argument) occurs inside a term (its second argument).
- Often, this check is skipped:

```
bool check (ref u, ref v) {  
    return true;  
}
```

Unification

Otherwise, we could implement check() as follows:

```
bool check (ref u, ref v) {
    if (u == v) return false;
    if (H[v] == (S,f/n))
        for (int i=1; i<=n; i++)
            if (!check (u, deref (H[v+i])))
                return false;
    return true;
}
```

Unification

- The translation of an equation $\tilde{X} = t$ is very simple,
- but all the objects constructed to represent t which have corresponding matching object reachable from X becomes immediately *garbage*.
- **Idea:**
 - Push a reference to the run-time binding of \tilde{X} onto the stack.
 - Avoid construction of subterms of t as long as possible.
 - Instead, translate each node of t into an instruction which performs the unification with this node!

$$\text{code}_G (\tilde{X} = t) \rho = \begin{array}{l} \text{put } \tilde{X} \rho \\ \text{code}_U t \rho \end{array}$$

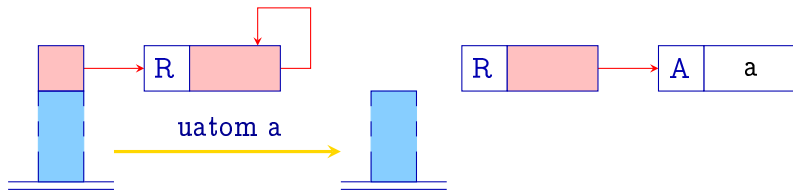
Unification

Unification of atoms and variables:

$\text{code}_U a \rho = \text{uatom } a$
 $\text{code}_U X \rho = \text{uvar } (\rho X)$
 $\text{code}_U \bar{X} \rho = \text{uref } (\rho X)$
 $\text{code}_U - \rho = \text{pop}$

Unification

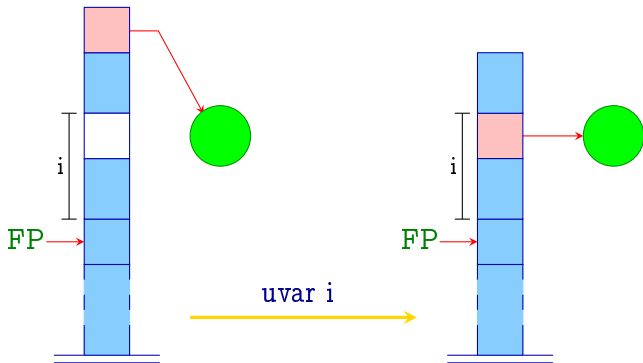
Instruction `uatom a` implements the unification with an atom:



```
v = S[SP]; SP--;  
switch (H[v]) {  
  case (A,a): break;  
  case (R,_): H[v] = (R, new(A,a));  
              trail(v); break;  
  default:   backtrack();  
}
```

Unification

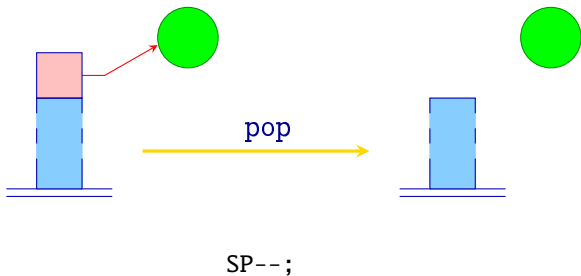
Instruction `uvar i` implements the unification with an uninitialized variable:



```
S[FP+i] = S[SP];  
SP--;
```

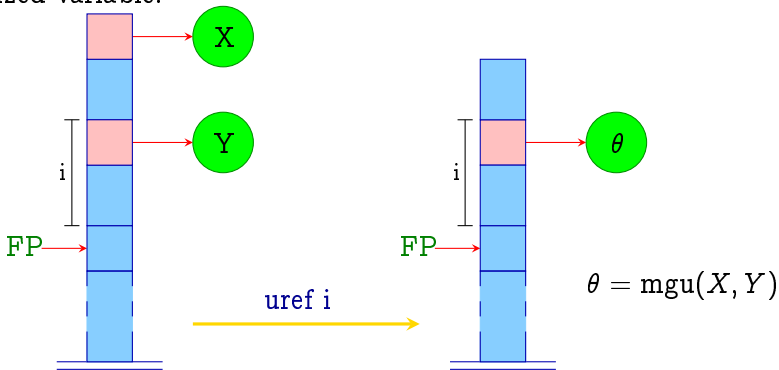
Unification

Instruction `pop` implements the unification with an anonymous variable:



Unification

Instruction `uref i` implements the unification with an initialized variable:



```
unify (S[SP], deref(S[FP+i]));  
SP--;
```

The only place, where the run-time function `unify()` is called!

Unification

Unification of constructor applications:

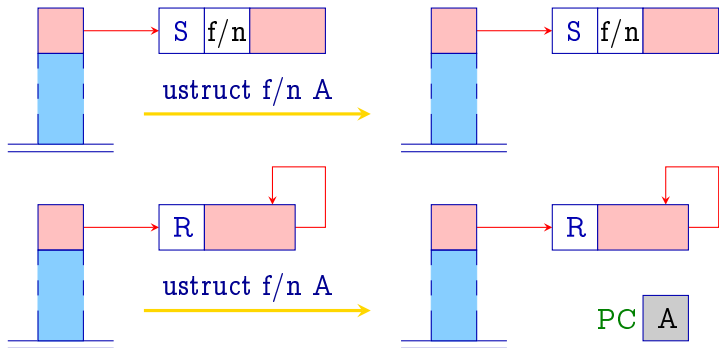
- The unification code performs a *pre-order* traversal over t .
- First it checks whether the root node is unifiable.
- If both terms have the same topmost constructor, then recursively checks for subterms.
- In the case of an uninitialized variable switches from checking to building.

Unification

Unification of constructor applications:

<code>code_U</code> ($f(t_1, \dots, t_n)$) ρ =	
<code>ustruct</code> f/n A	<code>up</code> B
<code>son</code> 1	A: <code>check</code> ivars($f(t_1, \dots, t_n)$) ρ
<code>code_U</code> t_1 ρ	<code>code_A</code> ($f(t_1, \dots, t_n)$) ρ
...	<code>bind</code>
<code>son</code> n	B: ...
<code>code_U</code> t_n ρ	

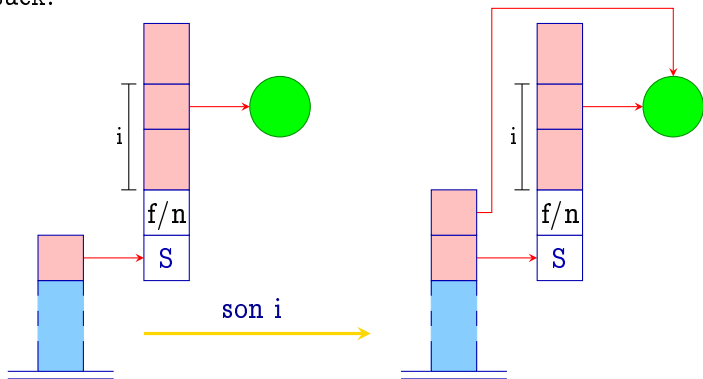
Unification



```
switch (H[S[SP]]) {  
  case (S,f/n):  break;  
  case (R,_):   PC = A; break;  
  default:      backtrack();  
}
```

Unification

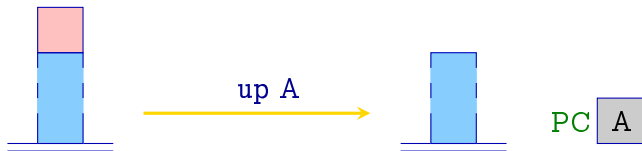
Instruction `son i` pushes the reference of the i -th subterm onto the stack:



```
S[SP+1] = deref (H[S[SP]]+i);  
SP++;
```

Unification

Instruction `up A` pops a reference from the stack and jumps to the continuation address:



```
SP--;  
PC = A;
```

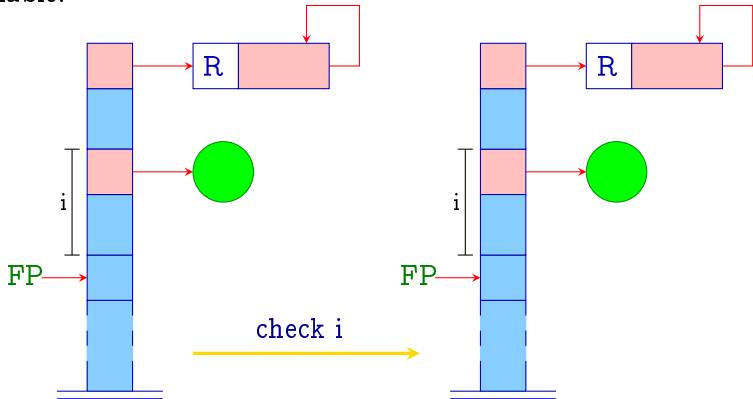
Unification

- In the case of an uninitialized variable we need to switch from checking to building.
- Before constructing the new term we need to exclude that it contains the variable on top of the stack:
 - the function `ivars(t)` returns the set of initialized variables of *t*;
 - the macro `check {Y1, ..., Yd} ρ` generates the necessary tests:

$$\text{check } \{Y_1, \dots, Y_d\} \rho = \begin{array}{l} \text{check } (\rho Y_1) \\ \dots \\ \text{check } (\rho Y_d) \end{array}$$

Unification

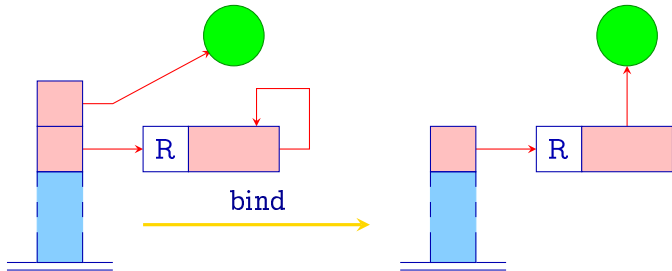
Instruction `check i` tests whether the (uninitialized) variable on top of the stack occurs inside the term bound to the i -th variable:



```
if (!check (S[SP], deref(S[FP+i])))  
    backtrack();
```

Unification

Instruction **bind** binds the (uninitialized) variable to the constructed term:



```
H[S[SP-1]] = (R, S[SP]);  
trail (S[SP-1]);  
SP = SP - 2;
```

Unification

Example: Let $t \equiv f(g(\bar{X}, Y), a, Z)$ with environment $\rho = \{X \mapsto 1, Y \mapsto 2, Z \mapsto 3\}$. Then $\text{code}_U t \rho$ generates the code:

ustruct f/3 A ₁	putref 1	A ₁ : check 1
son 1	putvar 2	putref 1
ustruct g/2 A ₂	putstruct g/2	putvar 2
son 1	bind	putstruct g/2
uref 1	B ₂ : son 2	putatom a
son 2	uatom a	putvar 3
uvar 2	son 3	putstruct f/3
up B ₂	uvar 3	bind
A ₂ : check 1	up B ₁	B ₁ : ...

Clauses

The code for clauses will:

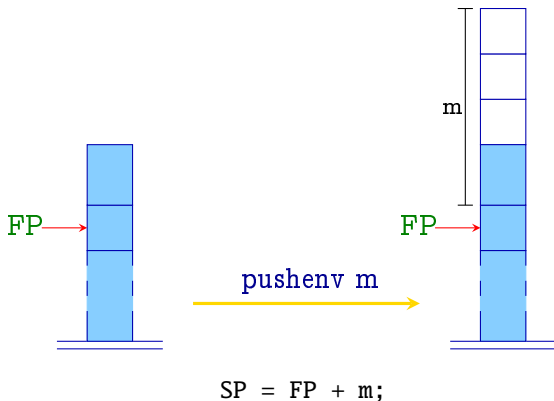
- allocate stack space for locals;
- evaluate the body;
- free the stack frame (if possible).

We denote local variables by $\{X_1, \dots, X_m\}$, where the first k ones are formal parameters.

$$\text{code}_G (p(X_1, \dots, X_k) \leftarrow g_1, \dots, g_n) = \begin{array}{l} \text{pushenv } m \\ \text{code}_G g_1 \rho \\ \dots \\ \text{code}_G g_n \rho \\ \text{popenv} \end{array}$$

Clauses

Instruction `pushenv m` allocates stack space for local variables:



Clauses

Example: Let

$$r \equiv a(X, Y) \leftarrow f(\bar{X}, X_1), a(\bar{X}_1, \bar{Y})$$

Then `codeC` r generates the code:

pushenv 3	call f/2	putref 2
mark A	A: mark B	call a/2
putref 1	putref 3	B: popenv
putvar 3		

Predicates

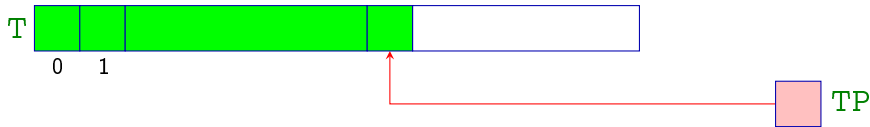
- A predicate q/k is defined by a sequence of clauses
 $rr \equiv r_1 \dots r_f$.
- The translation of predicates is performed by the function code_P .
- If a predicate has just a single clause (ie. $f = 1$), we have:
$$\text{code}_P r = \text{code}_C r$$
- If a predicate has several clauses, then:
 - we first "try" the first clause;
 - if it fails, then "try" the second one; etc.

Predicates

- If unification fails, we call the run-time function `backtrack()`.
- The goal is to roll back the whole computation to the *bactrack point*; ie. to the (dynamically) latest goal where there is another clause to "try".
- In order to restore previously valid bindings, we have used the run-time function `trail()` which stores new bindings in a special memory area.

Predicates

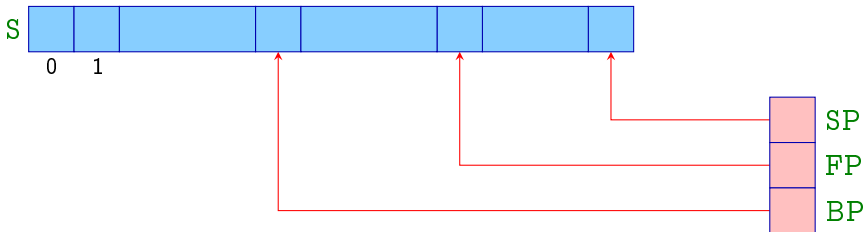
Trail:



T = Trail — memory area for storing new bindings;
TP = Tail-Pointer — points to the topmost used cell.

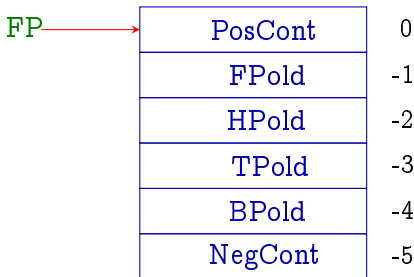
Predicates

There is also a special register **BP** which points to the current backtrack point.



Predicates

A **backtrack point** is a stack frame to which program execution possibly returns:



We will use following macros to denote organizational cells:

PosCont	≡	S[FP]	TPold	≡	S[FP-3]
FPold	≡	S[FP-1]	BPold	≡	S[FP-4]
HPold	≡	S[FP-2]	NegCont	≡	S[FP-5]

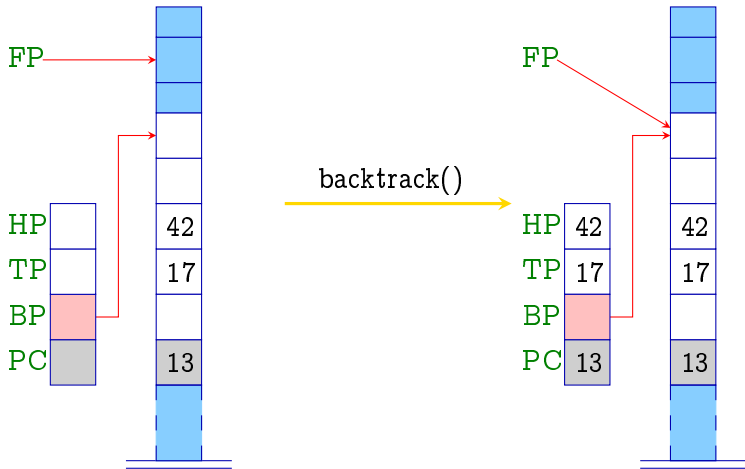
Predicates

The run-time function `backtrack()` restores registers according to the frame corresponding to backtrack point:

```
void backtrack() {  
    FP = BP;  
    HP = HPold;  
    reset (TPold,TP);  
    TP = TPold;  
    PC = NegCont;  
}
```

The function `reset()` restores variable bindings; ie. undoes all bindings created after the backtrack point.

Predicates



Predicates

- The variables which are created since the last backtrack point can be removed together with their bindings simply by restoring the old value of the register **HP**.
- This works fine if *younger* variables always point to *older* objects.
- Bindings where *older* variables point to *younger* objects must be reset "manually".
- These bindings are recorded in the *trail*.

Predicates

The function `trail()` records a binding if the argument points to a younger object:

```
void trail (ref u) {
    if (u < S[BP-2]) {
        TP = TP+1;
        T[TP] = u;
    }
}
```

The cell `S[BP-2]` contains the value of `HP` before the creation of backtrack point.

The function `reset()` removes all bindings created after the last backtrack point:

```
void reset (ref x, ref y) {
    for (ref u=y; x<u; u--)
        H[T[u]] = (R,T[u]);
}
```

Predicates

Translation of a predicate q/k , which is defined by clauses r_1, \dots, r_f ($f > 1$), generates a code which:

- creates a backtrack point;
- successively "tries" the alternatives;
- deletes the backtrack point.

Predicates

```
codeP (r1, ..., rf) =  q/k: setbtp           jump Af
                        try A1           A1: codeC r1
                        ...
                        try Af-1       Af: codeC rf
                        delbtp
```

NB!

- The backtrack point is deleted before the last alternative is "tried".
- For the "last try", the code jumps directly to the alternative and never returns to the present frame.

Predicates

Example:

$$\begin{aligned} s(X) &\leftarrow t(\bar{X}) \\ s(X) &\leftarrow \bar{X} = a \end{aligned}$$

Translation of the predicate `s/1` results:

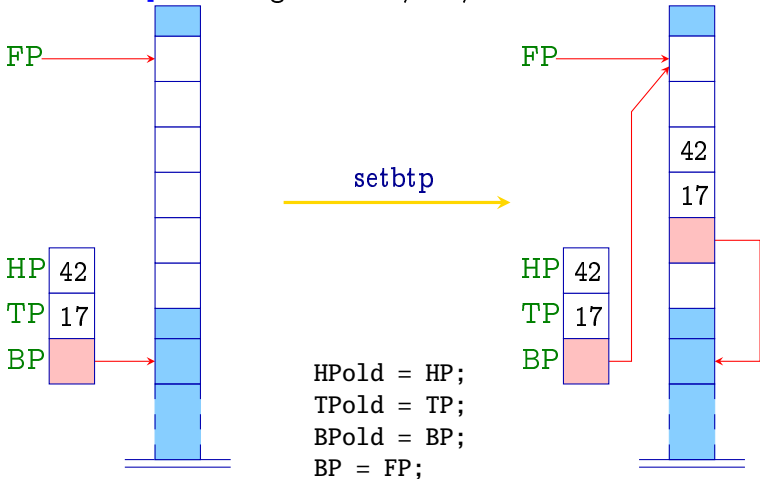
```
s/1: setbtp  
     try A  
     delbtp  
     jump B
```

```
A: pushenv 1  
   mark C  
   putref 1  
   call t/1  
C: popenv
```

```
B: pushenv 1  
   putref 1  
   uatom a  
   popenv
```

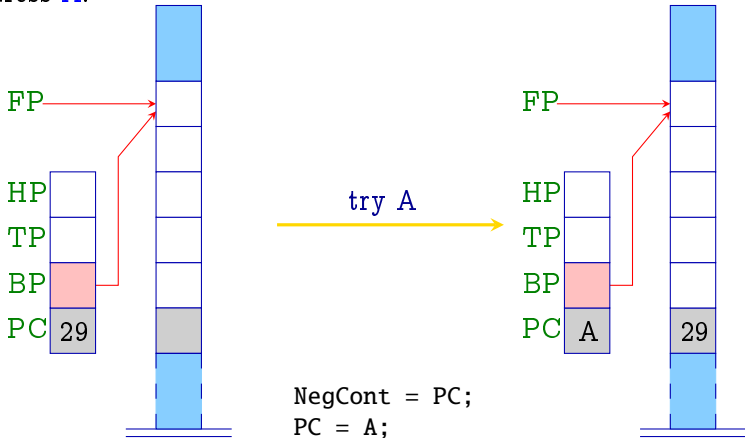
Predicates

Instruction `setbtp` saves registers `HP`, `TP`, `BP`:



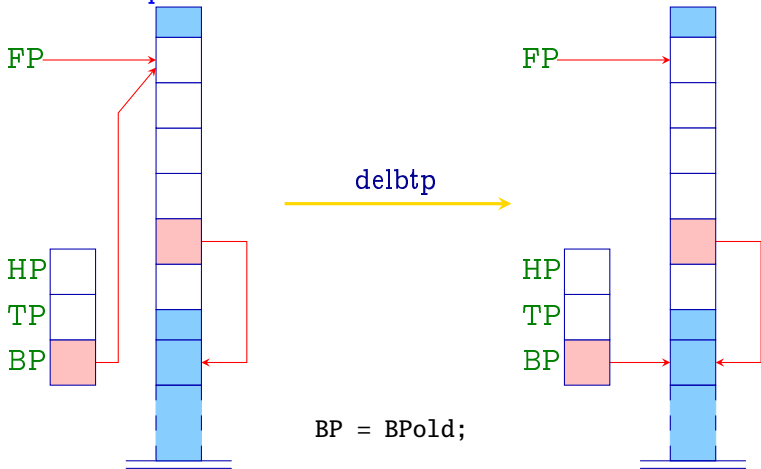
Predicates

Instruction `try A` saves the current `PC` as the negative continuation address and jumps to the alternative to be "tried" at address `A`:



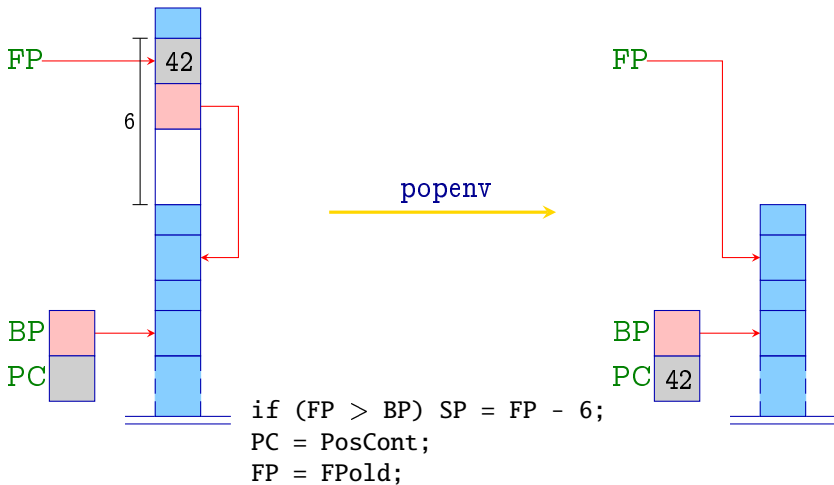
Predicates

Instruction `delbtp` restores the value of `BP`:



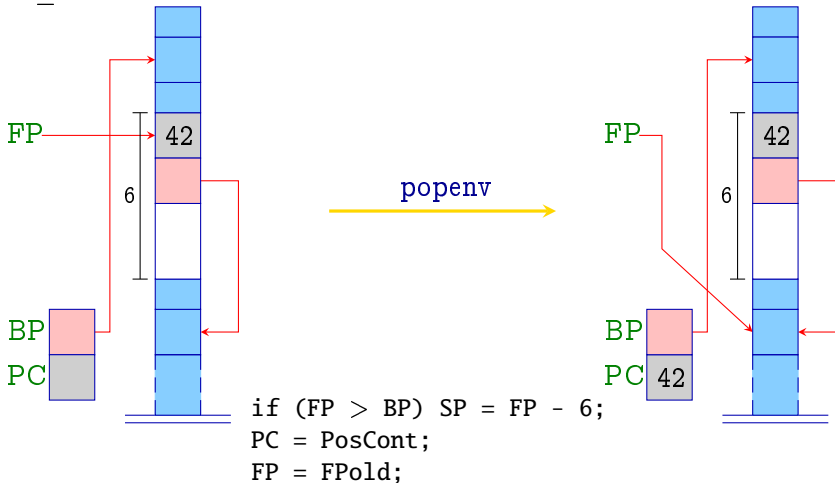
Predicates

Instruction `popenv` restores registers `FP` and `PC`, and if possible pops the stack frame:



Predicates

If $FP \leq BP$ the the frame is not deallocated:



Queries and Programs

- Translation of a program $p \equiv rr_1 \dots rr_h?g$ generates:
 - code for evaluating the query g ;
 - code for the predicate definitions rr_i .
- Query evaluation is preceded by:
 - initialization of registers;
 - allocation of space for globals.
- Query evaluation is succeeded by:
 - returning the values of globals.

Queries and Programs

```
code (rr1 ... rrh ?g) =  init           codeP rr1
                          pushenv d      ...
                          codeG g ρ      codeP rrh
                          halt d
```

where $free(g) = \{X_1, \dots, X_d\}$ and $\rho = \{X_i \mapsto i \mid i = 1 \dots d\}$.

Instruction `halt d ...`

- ... terminates the program execution;
- ... returns the values of d globals;
- ... if user requests, performs backtracking.

Queries and Programs

Instruction `init` creates the initial backtrack point:

FP	-1
HP	0
TP	-1
BP	-1

`init`

FP		
HP	0	
TP	-1	0
BP		-1
		-1
		f

`BP = FP = SP = 5;`
`S[0] = f;`
`S[1] = S[2] = -1;`
`S[3] = 0;`
`BP = FP;`

If the query `g` fails, the code at address `f` will be executed (eg. prints a message telling about the failure).

Queries and Programs

Example:

$$\begin{array}{lll} t(X) \leftarrow \bar{X} = b & q(X) \leftarrow s(\bar{X}) & s(X) \leftarrow \bar{X} = a \\ p \leftarrow q(X), t(\bar{X}) & s(X) \leftarrow t(\bar{X}) & ? p \end{array}$$

init	p/0: pushenv 1	q/1: pushenv 1	E: pushenv 1
pushenv 0	mark B	mark D	mark G
mark A	putvar 1	putref 1	putref 1
call p/0	call q/1	call s/1	call t/1
A: halt 0	B: mark C	D: popenv	G: popenv
t/1: pushenv 1	putref 1	s/1: setbtp	F: pushenv 1
putref 1	call t/1	try E	putref 1
uatom b	C: popenv	delbtp	uatom a
popenv		jump F	popenv

Last Call Optimization

Consider the predicate `app/3` defined as follows:

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

The last goal of the second clause is a recursive call:

- we can evaluate it in the current stack frame;
- after (successful) completion, we will not return to the current frame but go directly back to the "predecessor" frame.

Last Call Optimization

Consider a clause $r \equiv p(X_1, \dots, X_k) \leftarrow g_1, \dots, g_n$, which has m local variables and where $g_n \equiv q(t_1, \dots, t_h)$.

<code>code_C r</code>	=	<code>pushenv m</code>	<code>code_A t₁ ρ</code>
		<code>code_G g₁ ρ</code>	<code>...</code>
		<code>...</code>	<code>code_A t_h ρ</code>
		<code>code_G g_{n-1} ρ</code>	<code>call q/h</code>
		<code>mark B</code>	<code>B: popenv</code>

Last Call Optimization

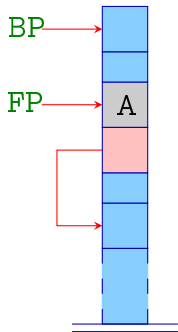
Consider a clause $r \equiv p(X_1, \dots, X_k) \leftarrow g_1, \dots, g_n$, which has m local variables and where $g_n \equiv q(t_1, \dots, t_h)$.

```
codeC r    =  pushenv m           codeA t1 ρ
               codeG g1 ρ         ...
               ...                 codeA th ρ
               codeG gn-1 ρ       lastcall (q/h,m)
               lastmark
```

Last Call Optimization

- If the current clause is not last or goals g_1, \dots, g_{n-1} have created backtrack points, then $FP \leq BP$.
- Then the instruction `lasmark` creates a new frame but stores a reference to the predecessor frame.
- Otherwise (ie. if $FP > BP$), it does nothing.

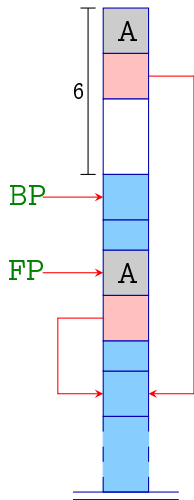
Last Call Optimization



lastmark



```
if (FP ≤ BP) {  
    SP = SP + 6;  
    S[SP] = PosCont;  
    S[SP-1] = FPold;  
}
```

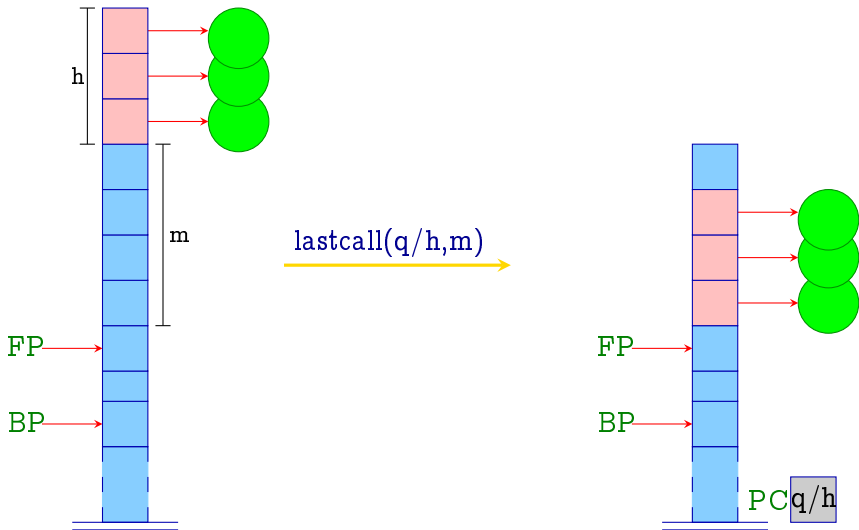


Last Call Optimization

- If $FP \leq BP$, then the instruction `lastcall (q/h,m)` behaves like `call q/h`.
- Otherwise, the current stack frame is reused:
 - the cells $S[FP+1], \dots, S[FP+h]$ get new values;
 - and then directly jumps to the predicate `q/h`.

```
lastcall (q/h,m) = if (FP ≤ BP) call q/h;
                  else {
                      move (m,h);
                      jump q/h;
                  }
```

Last Call Optimization



Last Call Optimization

Consider the clause

$$a(X, Y) \leftarrow f(\bar{X}, X_1), a(\bar{X}_1, \bar{Y})$$

The last call optimization yields:

```
pushenv 3          putvar 3          putref 3
mark A            call f/2          putref 2
putref 1          A: lastmark    lastcall(a/2,3)
```

NB! If the clause is last and its last goal is the only one, then we can omit `lastmark` and replace `lastcall(q/h,m)` with instructions `move(m,h)` and `jump q/h`.

Last Call Optimization

The last call optimization for the second clause of app/3 yields:

A: pushenv 6	putstruct []/2	D: check 4
putref 1	bind	putref 4
ustruct []/2 B	C: putref 3	putvar 6
son 1	ustruct []/2 D	putstruct []/2
uvar 4	son 1	bind
son 2	uref 4	E: putref 5
uvar 5	son 2	putref 2
up C	uvar 6	putref 6
B: putvar 4	up E	move(6,3)
putvar 5		jump app/3

Stack Frame Trimming

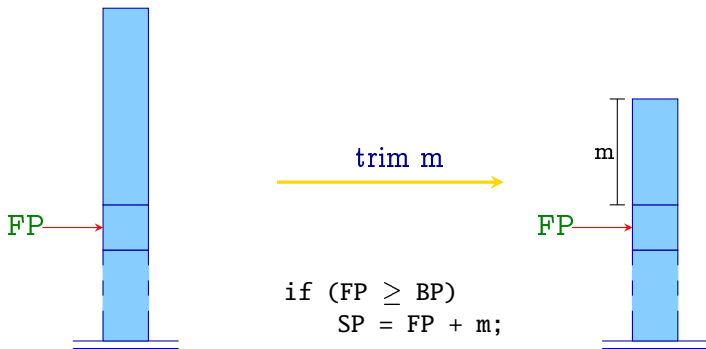
- Order local variables according to their *life time*.
- If possible, remove *dead* variables.
- Example:

$$a(X, Z) \leftarrow p_1(\bar{X}, X_1), p_2(\bar{X}_1, X_2), p_3(\bar{X}_2, X_3), p_4(\bar{X}_3, Z)$$

- after the goal $p_2(\bar{X}_1, X_2)$ the variable X_1 is dead;
- after the goal $p_3(\bar{X}_2, X_3)$ the variable X_2 is dead.

Stack Frame Trimming

After every non-last goal which has dead variables insert an instruction `trim`:



NB! We can remove dead locals only if there are no new backtrack points created.

Stack Frame Trimming

Example:

$$a(X, Z) \leftarrow p_1(\bar{X}, X_1), p_2(\bar{X}_1, X_2), p_3(\bar{X}_2, X_3), p_4(\bar{X}_3, Z)$$

Ordering of the variables:

$$\rho = \{X \mapsto 1, Z \mapsto 2, X_3 \mapsto 3, X_2 \mapsto 4, X_1 \mapsto 5\}$$

pushenv 5	putvar 4	C: trim 3
mark A	call p ₂ /2	lastmark
putref 1	B: trim 4	putref 3
putvar 5	mark C	putref 2
call p ₁ /2	putref 4	lastcall (p ₄ /2, 3)
A: mark B	putvar 3	
putref 5	call p ₃ /2	

Clause Indexing

- Often, predicates are implemented by case distinction on the first argument.
- Hence, by inspecting the first argument, many alternatives can be excluded.
 - Failure is detected earlier.
 - Backtrack points are removed earlier.
 - Stack frames are removed earlier.

Clause Indexing

Example:

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

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

- If the first argument is `[]`, then only the first clause is applicable.
- If the first argument has `[|]` as its root constructor, then only the second clause is applicable.
- Every other root constructor of the first argument will fail.
- Both alternatives should be tried only if the first argument is uninitialized variable.

Clause Indexing

- Introduce a separate *try chain* for every possible constructor.
- Inspect the root node of the first argument.
- Depending on the result, perform an indexed jump to the appropriate try chain.

Let the predicate *p/k* defined by the sequence of clauses

$$rr \equiv r_1 \dots r_m.$$

The macro **tchains** *rr* denotes the sequence of try chains which correspond to the root constructors occurring in unifications $X_1 = t$.

Clause Indexing

Example:

Consider the predicate `app/3`. Let the code for its two clauses start at addresses A_1 and A_2 . Then we get the following four try chains:

```
VAR: setbtb // variables      NIL: jump A1 // []
    try A1                  CONS: jump A2 // [[]]
    delbtp                    ELSE: fail // default
    jump A2
```

Instruction `fail` handles all constructors besides `[]` and `[[]]`.

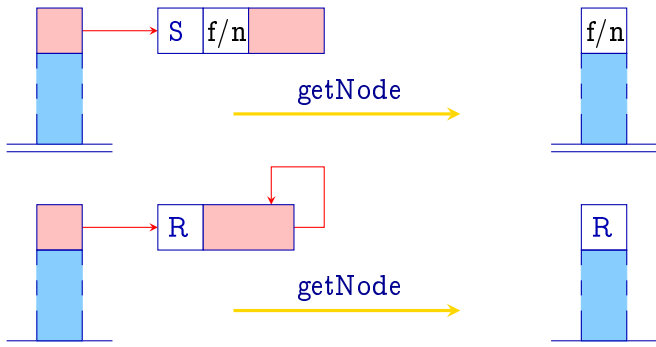
```
fail = backtrack()
```

Clause Indexing

Then we generate for a predicate p/k:

```
codeP rr =      putref 1  
                getNode  
                index p/k  
                tchains rr  
A1: codeC r1  
    ...  
Am: codeC rm
```

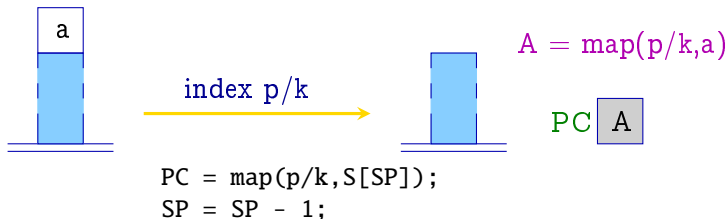
Clause Indexing



```
switch (H[S[SP]]) {  
  case (S,f/n):  S[SP] = f/n; break;  
  case (A,a):   S[SP] = a; break;  
  case (R,_):   S[SP] = R;  
}
```

Clause Indexing

Instruction `index p/k` performs an indexed jump to the appropriate try chain:



The function `map()` returns the start address of the appropriate try chain. Can be defined eg. through some hash table.

Cut Operator

We extend the language **Proll** with the cut operator "!" which explicitly allows to prune the search space of backtracking.

Example:

$$\begin{aligned} \text{branch}(X, Y) &\leftarrow p(X), !, q_1(X, Y) \\ \text{branch}(X, Y) &\leftarrow q_2(X, Y) \end{aligned}$$

If all the queries before the cut have succeeded, then the choice is *committed*: backtracking will return only to backtrack points *preceding* the call to the predicate.

Cut Operator

The cut operator should:

- restore the register `BP` by assigning to it `BPold` from the current frame;
- remove all frames which are on top of the local variables.

Accordingly, we translate the cut into the sequence:

```
prune  
pushenv m
```

where m is the number of (still alive) local variables of the clause.

Cut Operator

Example:

branch(X, Y) \leftarrow p(X), !, q₁(X, Y)
branch(X, Y) \leftarrow q₂(X, Y)

We obtain:

setbtp	A: pushenv 2	C: prune	B: pushenv 2
try A	mark C	pushenv 2	putref 1
delbtp	putref 1	lastmark	putref 2
jump B	call p/1	putref 1	move(2,2)
		putref 2	jump q ₂ /2
		lastcall(q ₁ /2,2)	

Cut Operator

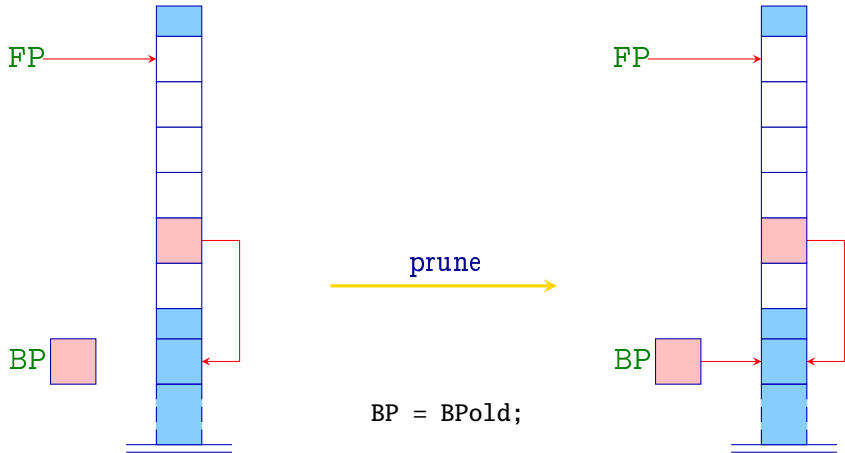
Example:

branch(X, Y) \leftarrow p(X), !, q₁(X, Y)
branch(X, Y) \leftarrow q₂(X, Y)

... or, when using optimizations:

setbtp	A: pushenv 2	C: prune	B: pushenv 2
try A	mark C	pushenv 2	putref 1
delbtp	putref 1	putref 1	putref 2
jump B	call p/1	putref 2	move(2,2)
		move(2,2)	jump q ₂ /2
		jump q ₁ /2	

Cut Operator



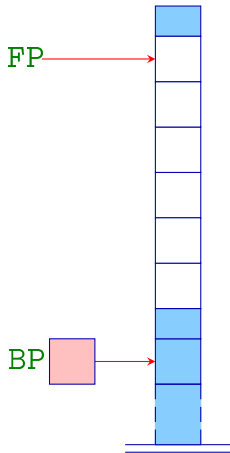
Cut Operator

Problem:

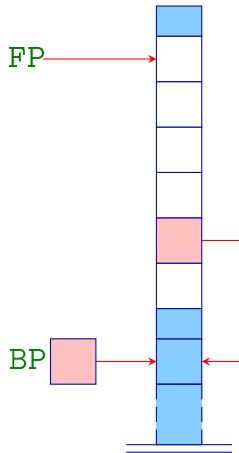
If the predicate is defined by a *single* clause, then we have not stored the old BP inside the stack frame.

For the cut to work also with single-clause predicates or try chains of length 1, we insert an extra instruction `setcut` before the clausal code (or the jump).

Cut Operator



BPold = BP;



Cut Operator

Final example: the predicate `notP` succeeds whenever `p` fails and vice versa:

```
notP(X) ← p(X), !, fail
notP(X) ←
```

where `fail` always fails.

```
setbtp    A: pushenv 1    C: prune    B: pushenv 1
try A     mark C         pushenv 1    popenv
delbtp    putref 1      fail
jump B    call p/1      popenv
```