Script generated by TTT

Title: Seidl: Virtual_Machines (15.06.2015)

Date: Mon Jun 15 10:26:11 CEST 2015

Duration: 82:24 min

Pages: 42

_123

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.

A More Realistic Example

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

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.

A More Realistic Example

$$
\begin{array}{l}
\mathsf{app}([\,], Z, Z). \\
\mathsf{app}([H|X'], Y, [H|Z']) \quad :- \quad \mathsf{app}(X', Y, Z'). \\
?- \quad \mathsf{app}(X, [Y, c], [a, b, Z]).
\end{array}
$$

A More Realistic Example

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

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:

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

$_1, \ldots, g_r$

- 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:

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

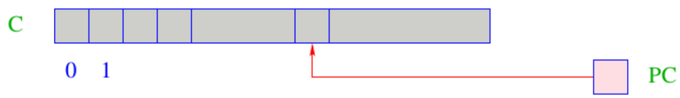
Note: Predicate calls ...

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

$\Longrightarrow$ backtracking

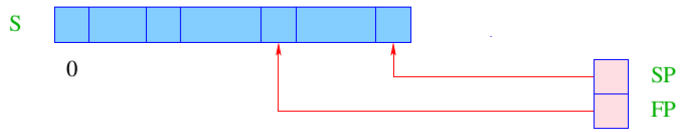
28 Architecture of the WiM:

The Code Store:



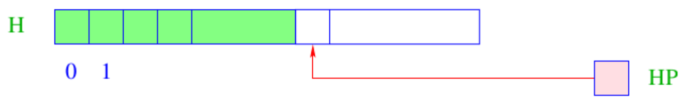
C	=	Code store – contains WiM program; every cell contains one instruction;
PC	=	Program Counter – points to the next instruction to executed;

The Runtime Stack:



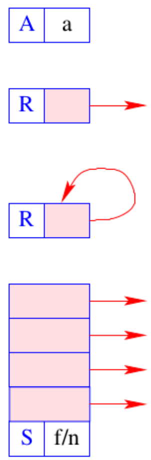
S	=	Runtime Stack – every cell may contain a value or an address;
SP	=	Stack Pointer – points to the topmost occupied cell;
FP	=	Frame Pointer – points to the current stack frame. Frames are created for predicate calls, contain cells for each variable of the current clause

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

29 Construction of Terms in the Heap

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

Assume that the address environment ρ returns, for each clause variable X its address (relative to FP) on the stack. Then $\boxed{\text{code}_A\ t\ \rho}$ 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 $\quad t \equiv f(g(X, Y), a, Z)$.

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

29 Construction of Terms in the Heap

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

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

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

Idea

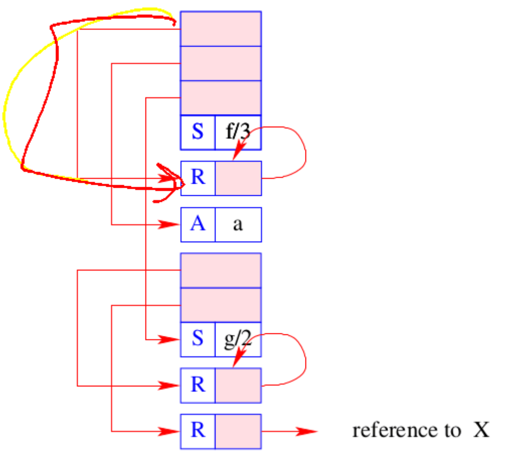
- Construct the tree during a post-order traversal of t
- with one instruction for each new node!

Example $\quad t \equiv f(g(X, Y), a, Z)$.

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

239

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



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:

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

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

Note: Arguments are always initialized!

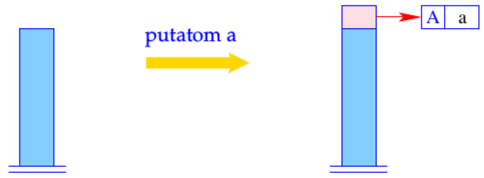
Then we define:

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

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

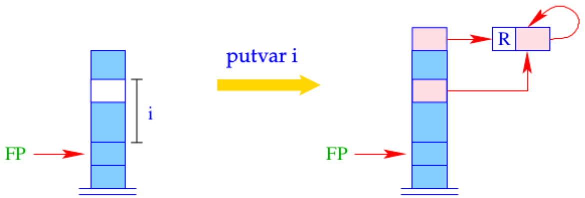
putref 1	putatom a
putvar 2	putvar 3
putstruct g/2	putstruct f/3

The instruction putatom a constructs an atom in the heap:



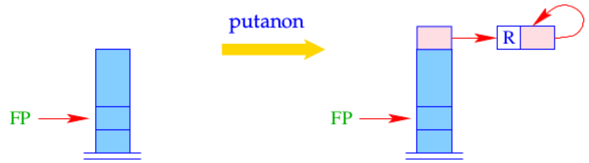
SP++; S[SP] = new (A,a);

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];

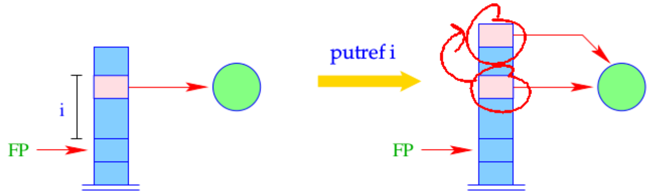
The instruction putanon introduces a new unbound variable but does not store a reference to it in the stack frame:



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

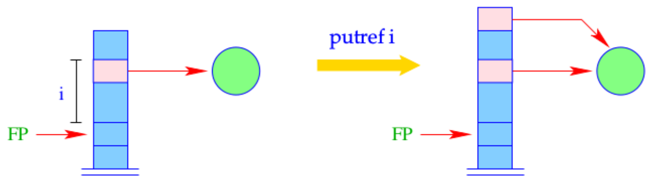
$f(\_, \_) = f(X, Y)$

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:

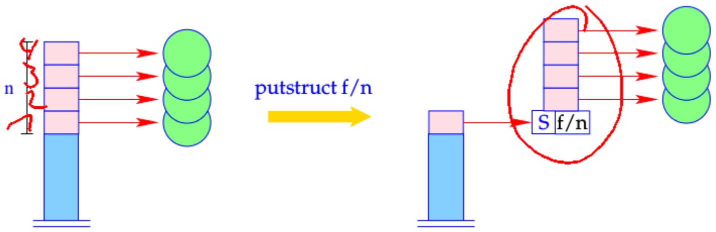


SP = SP + 1;
S[SP] = deref S[FP + i];

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 :-(

$p(t_1, t_2)$

30 The Translation of Literals

Idea

- Literals are treated as procedure calls.
- We first allocate a stack frame.
- Then we construct the actual parameters (in the heap)
- ... and store references to these into the stack frame.
- Finally, we jump to the code for the procedure/predicate.

$$\begin{array}{rll} \text{code}_G\ p(t_1,\ldots,t_k)\ \rho \quad = & \text{mark B} & \text{// allocates the stack frame} \\ & \text{code}_A\ t_1\ \rho & \\ & \ldots & \\ & \text{code}_A\ t_k\ \rho & \\ & \text{call p/k} & \text{// calls the procedure p/k} \\ \text{B}: & \ldots & \end{array}$$

$$\begin{array}{rll} \text{code}_G\ p(t_1,\ldots,t_k)\ \rho \quad = & \text{mark B} & \text{// allocates the stack frame} \\ & \text{code}_A\ t_1\ \rho & \\ & \ldots & \\ & \text{code}_A\ t_k\ \rho & \\ & \text{call p/k} & \text{// calls the procedure p/k} \\ \text{B}: & \ldots & \end{array}$$

Example $\quad p(a, X, g(\bar{X}, Y)) \quad$ with $\quad \rho = \{X \mapsto 1, Y \mapsto 2\}$

We obtain:

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

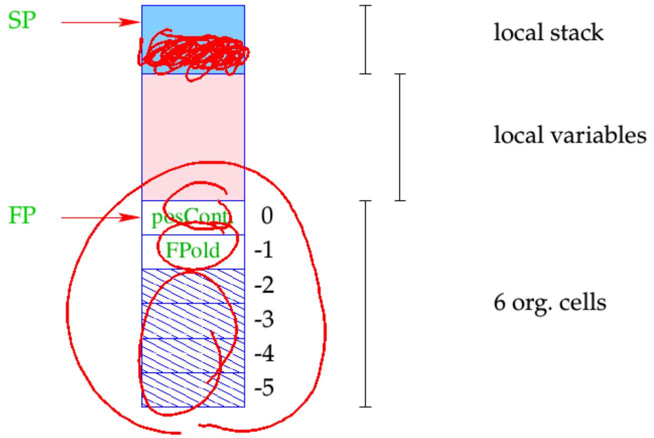
$$\begin{array}{rll} \text{code}_G\ p(t_1,\ldots,t_k)\ \rho \quad = & \text{mark B} & \text{// allocates the stack frame} \\ & \text{code}_A\ t_1\ \rho & \\ & \ldots & \\ & \text{code}_A\ t_k\ \rho & \\ & \text{call p/k} & \text{// calls the procedure p/k} \\ \text{B}: & \ldots & \end{array}$$

Example $\quad p(a, X, g(\bar{X}, Y)) \quad$ with $\quad \rho = \{X \mapsto 1, Y \mapsto 2\}$

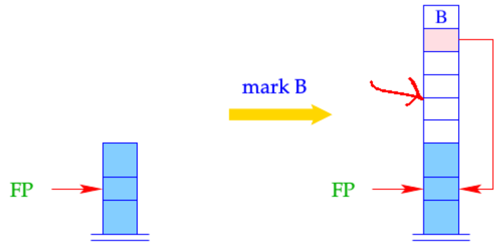
We obtain:

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

Stack Frame of the WiM:

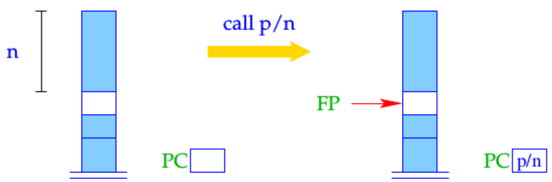


The instruction mark B allocates a new stack frame:



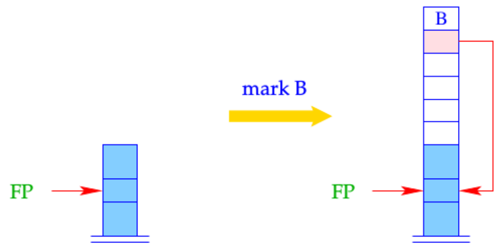
SP = SP + 6;
S[SP] = B; S[SP-1] = FP;

The instruction call p/n calls the n-ary predicate p :



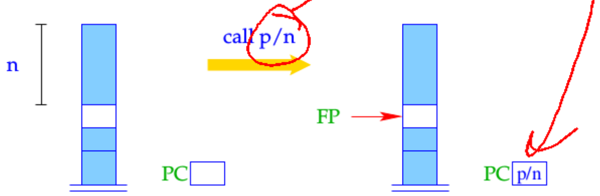
FP = SP - n;
PC = p/n;

The instruction mark B allocates a new stack frame:



SP = SP + 6;
S[SP] = B; S[SP-1] = FP;

The instruction call p/n calls the n-ary predicate p :



FP = SP - n;
PC = p/n;

31 Unification

Convention

- By $\tilde{X}$, we denote an occurrence of X; it will be translated differently depending on whether the variable is initialized or not.
- We introduce the macro put $\tilde{X}\ \rho$:

$$
\begin{array}{lll}
\text{put } X\ \rho & = & \text{putvar } (\rho\, X) \\
\text{put } \_\ \rho & = & \text{putanon} \\
\text{put } \bar{X}\ \rho & = & \text{putref } (\rho\, X)
\end{array}
$$

Let us translate the unification $\tilde{X} = t$.

Idea 1

- Push a reference to (the binding of) X onto the stack;
- Construct the term t in the heap;
- Invent a new instruction implementing the unification :-)

Let us translate the unification $\tilde{X} = t$.

Idea 1

- Push a reference to (the binding of) X onto the stack;
- Construct the term t in the heap;
- Invent a new instruction implementing the unification :-)

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

Example

Consider the equation:

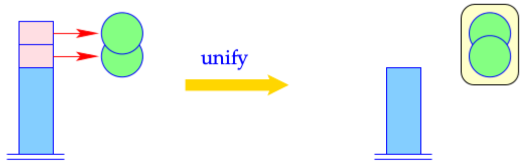
$$\bar{U} = \boxed{f(g(\bar{X}, Y), a, Z)}$$

Then we obtain for an address environment

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

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

The instruction unify calls the run-time function `unify()` for the topmost two references:




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


The Function `unify()`

- ... takes two heap addresses.
  For each call, we guarantee that these are maximally de-referenced.
- ... checks whether the two addresses are already identical.
  If so, does nothing :-)
- ... 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 $\Longrightarrow$ occur-check;
- ... records newly created bindings;
- ... may fail. Then backtracking is initiated.

Example

Consider the equation:

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

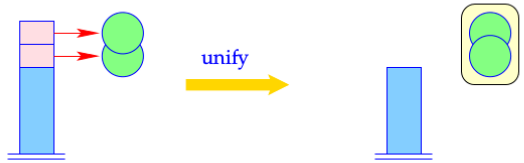
Then we obtain for an address environment

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

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

260

The instruction unify calls the run-time function `unify()` for the topmost two references:



unify (S[SP-1], S[SP]);
SP = SP–2;

The Function `unify()` $X = f(X)$

- ... takes two heap addresses.
  For each call, we guarantee that these are maximally de-referenced.
- ... checks whether the two addresses are already identical.
  If so, does nothing :-)
- ... 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 $\Longrightarrow$ occur-check;
- ... records newly created bindings;
- ... may fail. Then backtracking is initiated.

The Function `unify()`

- ... takes two heap addresses.
  For each call, we guarantee that these are maximally de-referenced.
- ... checks whether the two addresses are already identical.
  If so, does nothing :-)
- ... 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 $\Longrightarrow$ occur-check;
- ... records newly created bindings;
- ... may fail. Then backtracking is initiated.