



Limits of Computation

7 - A universal program (Self-interpreter)
Bernhard Reus

1



So far...

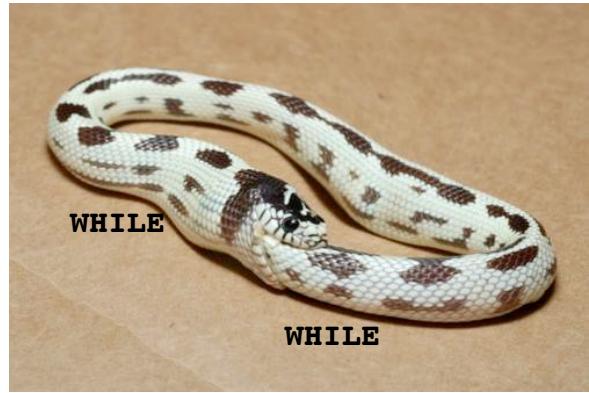
- ... we have learned the WHILE-language...
- ...that we have chosen to represent our notion of computation (to write “effective procedures”).
- We learned how to represent programs-as-data...
- ...so now we **can write interpreters.**

2



Eating your own tail?

- We look at a special form of interpreter: **THIS TIME**
- **self-interpreter**
 - WHILE-interpreter in WHILE
 - and first an WHILE-interpreter in WHILE
- a very important concept for computability theory (*used later*)



<http://www.strangedangers.com/content/item/158424.html>

3



Compare to TMs

- Turing defined a “universal Turing machine” U
- that can take TM program description D and a word W as input on its tape
- and simulate the run of TM D with given input W
- so U is a TM program which is an interpreter for TM programs

a self-interpreter in TM



let's use
WHILE instead!

4



Use of self-interpreter?

- in practice:
“cheap” way to extend your programming language with extra features (interpret them in smaller language)
- in computability theory:
we will explain this soon. Stay tuned!

5



First consider WH¹LE

- ...is like WHILE...
- ...but programs can only have **one** variable.
- simpler “memory management”
- Can we solve fewer problems in language WH¹LE than in WHILE?



6



Interpret WH¹LE in WHILE

- Since it is simpler, we first look at an interpreter of WH¹LE written in WHILE.
- Then we generalise to arbitrarily many variables and obtain a WHILE-interpreter in WHILE.

7



Tree Traversal of ASTs *(with intermediate results)*

NO
RECURSION !

initialise tree and value stack to be empty
push tree (to be traversed) on tree stack
while tree stack not empty
 pop a tree t from tree stack
 if t is just an opcode o with arity n // a marker
 then pop n results r_1, \dots, r_n from value stack
 $r := o(r_1, \dots, r_n)$ // compute intermediate result
 push r on value stack
 else // t proper tree
 if t 's opcode has n arguments
 then push t 's opcode on tree stack // (as marker!)
 push n subtrees of t on tree stack
 else // o is leaf
 compute result and push on value stack

order of
arguments
important

8

WHILE-interpreter

```

read PD {                                (* input is a list [P,D] *)
    P := hd PD ;                      (* P = [X,B,X] *)
    D := hd tl PD;                   (* D input data *)
    B := hd tl P;                    (* B is program block *)
    CSt := B;                         (* CSt is code stack *)
                                         (* initially commands of B *)
    DSt := nil;                       (* DSt is computation stack for *)
                                         (* intermediate results *)
    val := D;                          (* D is initial value of variable *)
    state := [ CSt, DSt, val ];       (* wrap up state for STEP macro *)
while CSt {                               (* main loop for interpretation *)
    state := <STEP> state;(* loop body macro *)
    CSt := hd state      (* get command stack *)
}
    val := hd tl tl state (* get final value of variable *)
}
write val                                 (* return value of the one variable *)

```

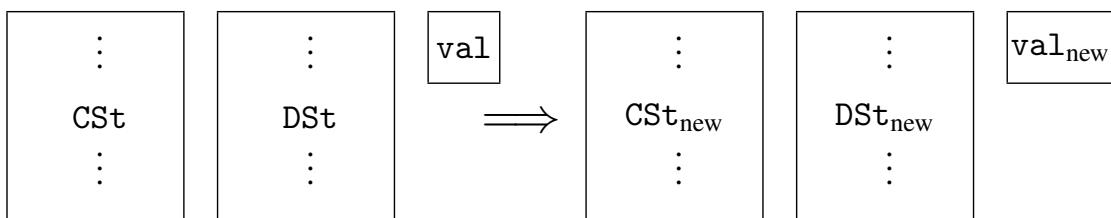
CSt is code stack (code in list format),
DSt is Stack of intermediate values,
val contains value D of the one variable

9

Step Macro

performs tree traversal based on CSt, DSt, and val.

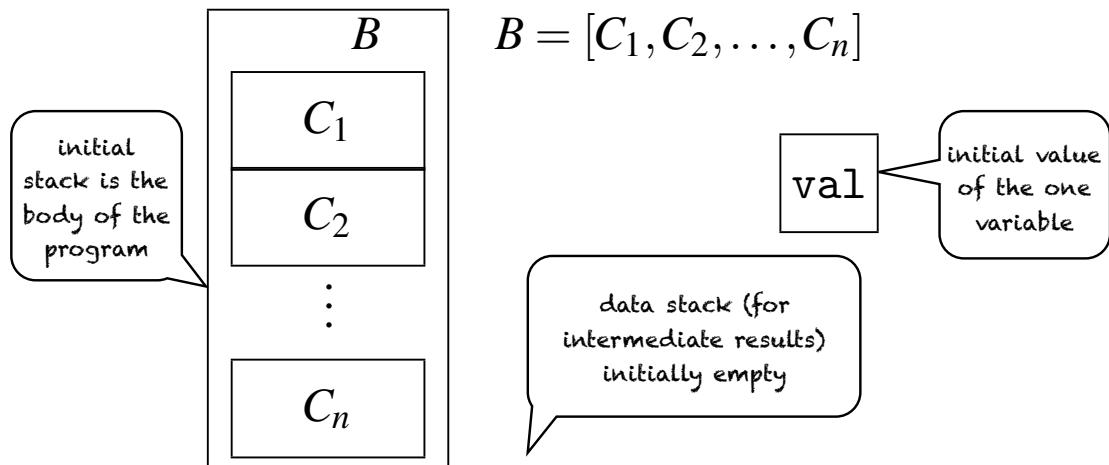
$$[CSt, DSt, val] \Rightarrow [CSt_{new}, DSt_{new}, val_{new}]$$





Initial set-up

state := [CSt, DSt, val];



11

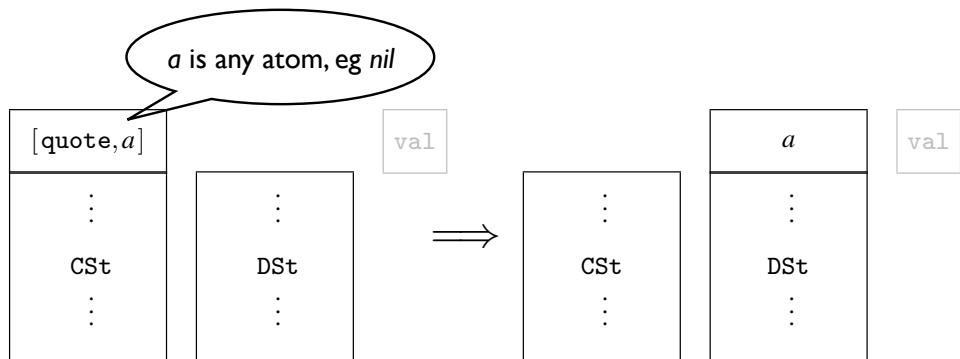


AST Leaves

(expressions without arguments)

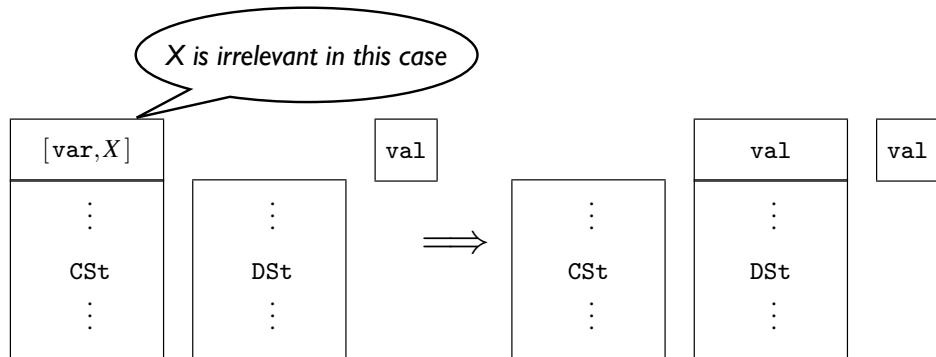
12

Atoms



13

Variable



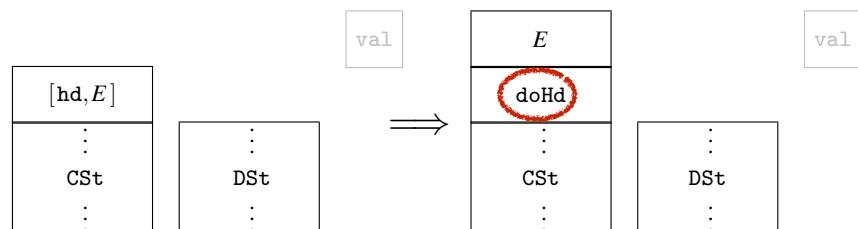
14

Compound Expressions

(unary and binary)

15

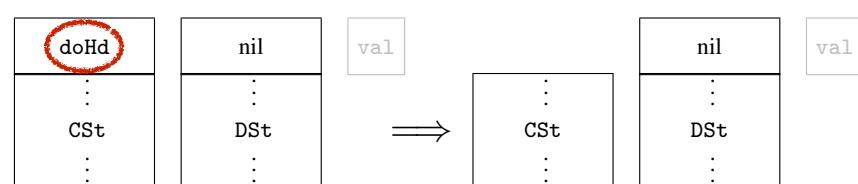
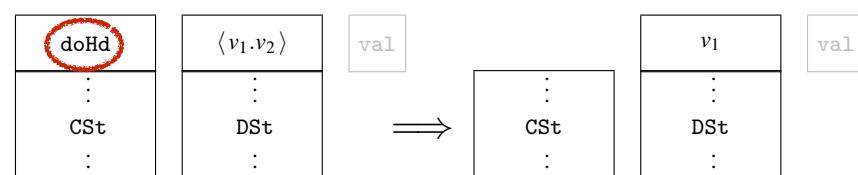
hd



(similarly for t1)

additional atoms
used as mnemonic

markers: what
needs to be done
when this marker
is popped off the
stack



16



Auxiliary atoms

We now add new (encoded) atoms to \mathbb{D}

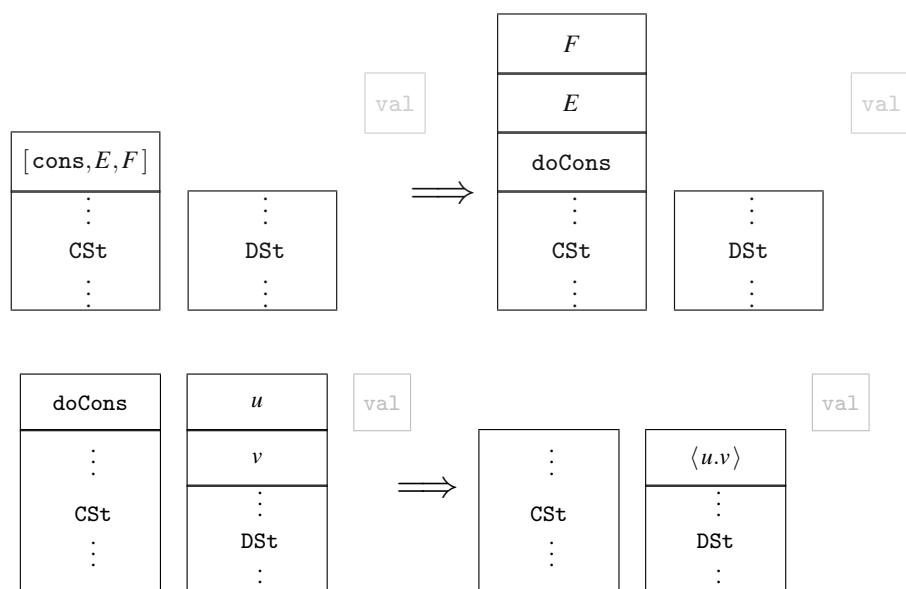
doHd, doTl, doCons, doAsgn, doIf, doWhile

Use: push on stack to indicate operation still to be done

17



cons



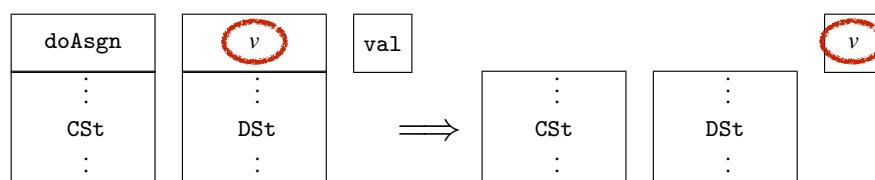
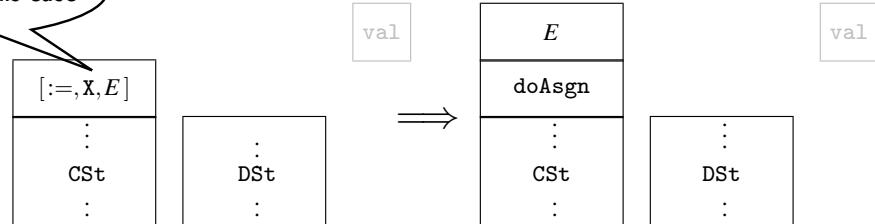
18

Commands

19

Assignment

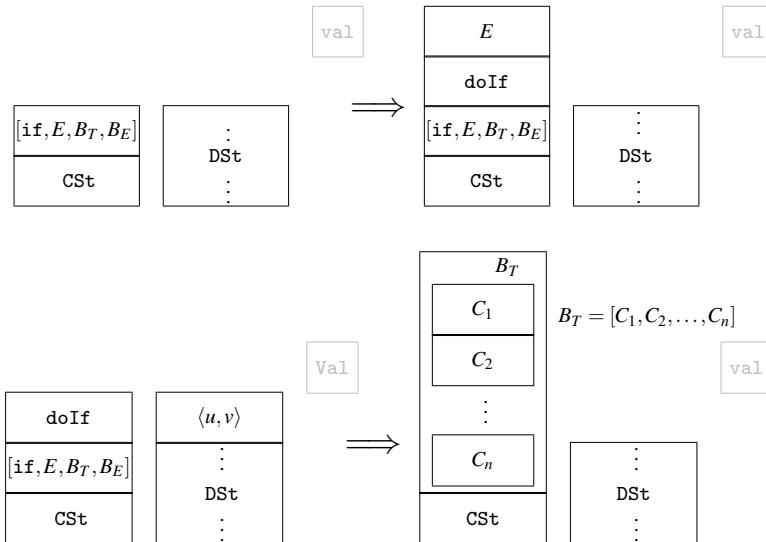
X is irrelevant in this case



20



if

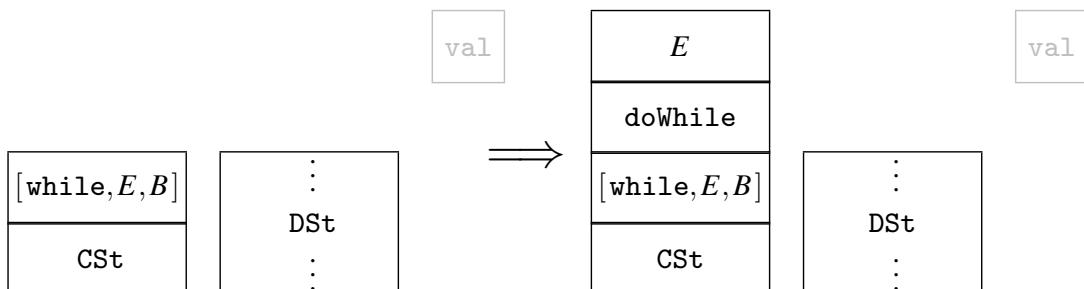


Analogously, if top element of DSt is *nil*, B_E is pushed on CSt

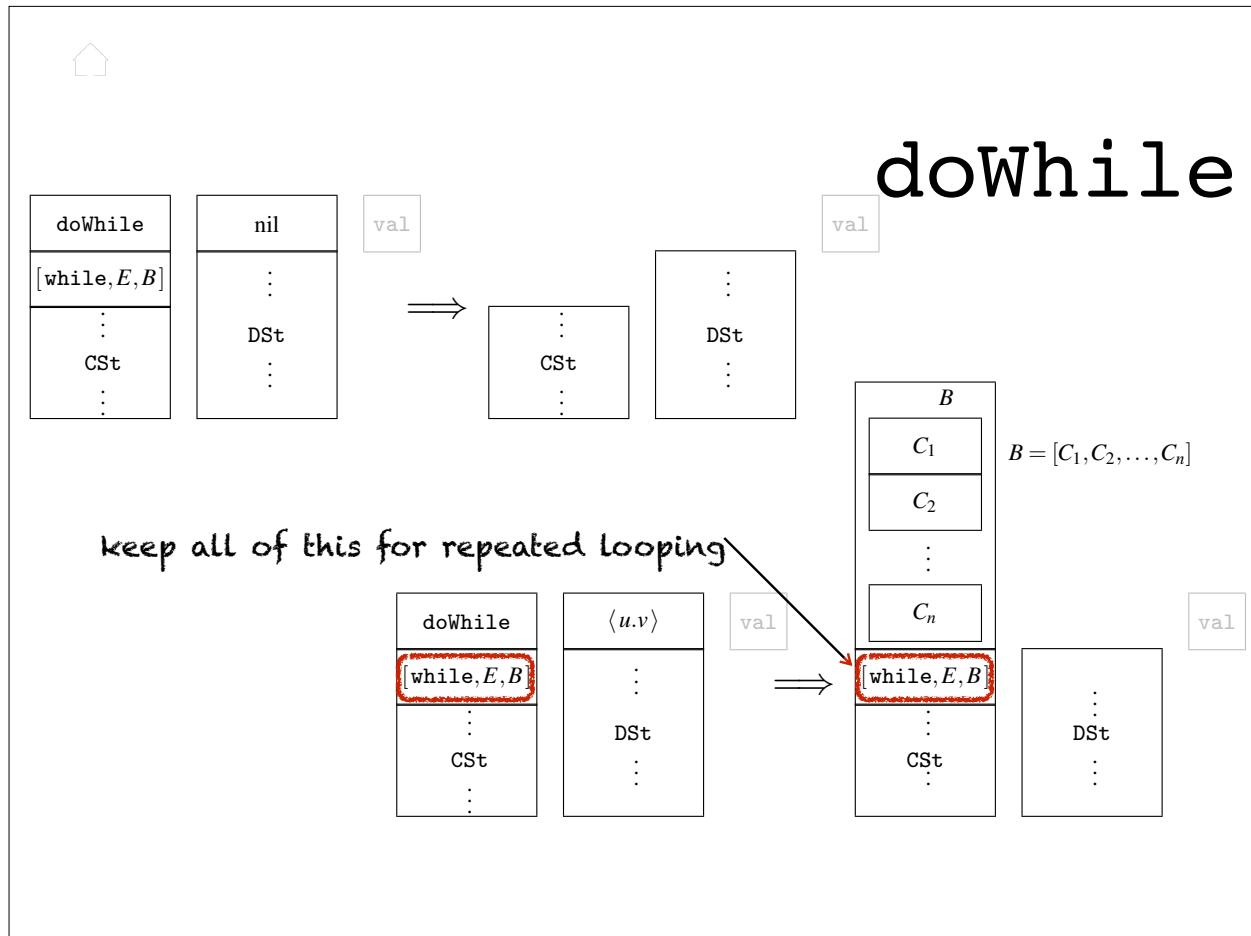
21



while



22

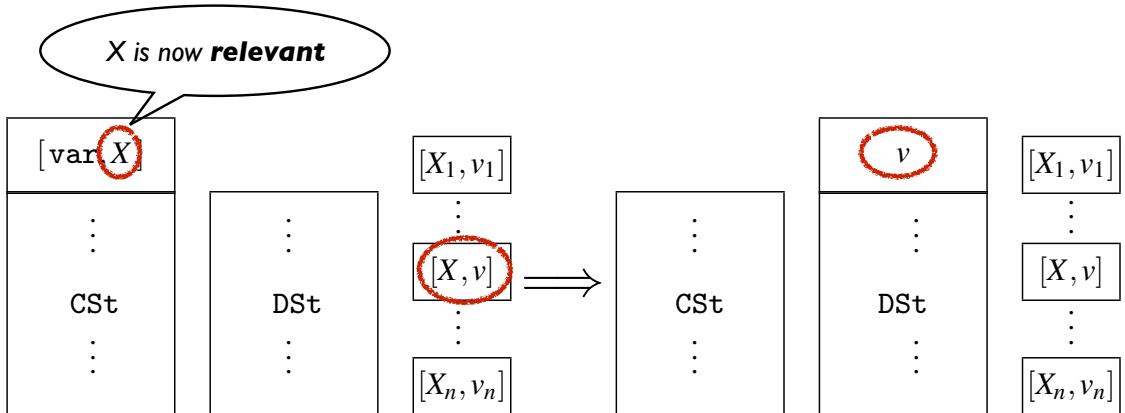


23

Changes to interpret WHILE



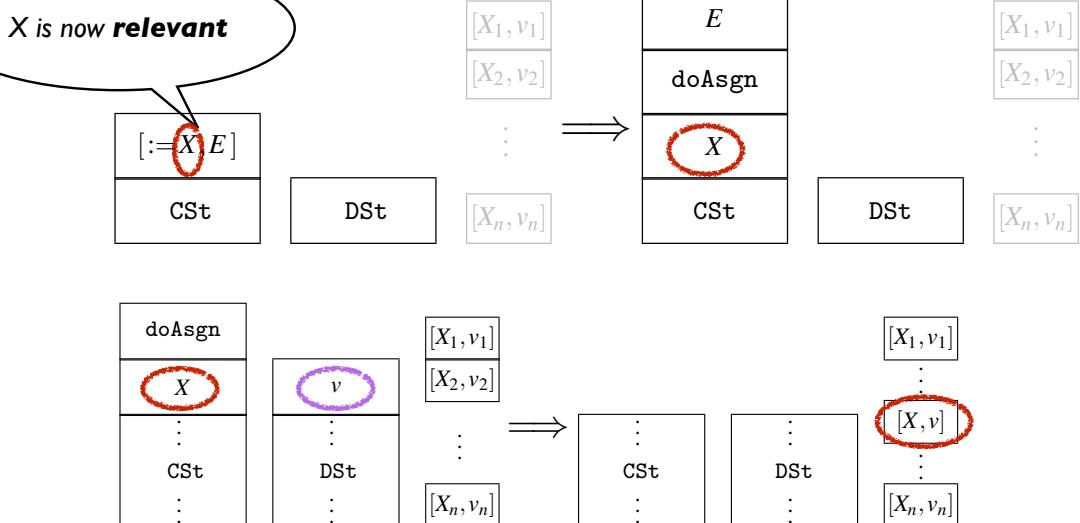
Variable lookup



25



Assignment



26

WHITE-interpreter

```

read PD {                                (* input is a list [X, D] *)
    P := hd PD ;
    D := hd tl PD;
    X := hd P;                          (* X is input var name *)
    Y := hd tl tl P;                   (* Y is output var name *)
    B := hd tl P;                      (* B is program code block *)
    CSt := B;                           (* CSt is code stack *)
    DST := nil;                         (* DST is data stack for *)
                                         (* intermediate results *)
}

bind := [ X, D ];
St := [ bind ];                         (* initialise store *)
state := [ CSt, DST, St ];              (* wrap state for STEP macro *)
while CSt {
    state := <STEPn> state;          (* loop body macro *)
    CSt := hd state                  (* get command stack *)
};
St := hd tl tl state;                  (* get final store *)
arg := [ Y, St ];                      (* wrap argument for lookup *)
Out := <lookup> arg;                  (* lookup output variable value *)
}

write Out                               (* return value of result variable *)

```

Cd is code stack (code in list format),
 Vl contains value d of the one variable,
 St is Stack of intermediate values

27

Coding the macros

- The update and lookup macro are available from Canvas as is the main interpreter loop.
- The STEP macro we might do partially at least in the exercises





END

© 2008-20. Bernhard Reus, University of Sussex

Next time: Our
first non-computable
problem